THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR ...Antony G. Olekshy...

TITLE OF THESIS ...Towards an Array Theoretic Functional Language:...

...The Artful Encoding of Redundancy...

...

DEGREE FOR WHICH THESIS WAS PRESENTED ...M. Sc....

YEAR THIS DEGREE GRANTED ...1980...

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's permission.

THE UNIVERSITY OF ALBERTA

Towards an Array Theoretic Functional Language: The Artful Encoding of Redundancy.

by

© Antony Gene Olekshy

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1980

THE UNIVERSITY OF ALBERTA

FACUTLY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "Towards an Array Theoretic Functional Language: The Artful Encoding of Redundancy." submitted by Antony Gene Olekshy, in partial fulfilment of the requirements for the degree of Master of Science in Computing Science.

Abstract

The use of computers as interactive problem-solving tools is increasing. In such an environment, efficient production algorithms are not usually required. Instead, the problem-solver requires a sophisticated work-bench on which to experiment with his problem and potential solutions. The primary requirement of such an environment is the ability to converse with the problem-solver on a very high level.

In addition, many attempts at developing very large systems are failing because of the psychological complexity of the programs that are being developed. Some of the features of the problem-solvers work-bench and some of the mechanisms for the reduction of the psychological complexity of very large systems are known. However, it is apparent that these features and mechanisms are themselves too psychologically complex to be successfully implemented with the facilities already available.

Instead, it will be necessary to bootstrap a hierarchy of more and more sophisticated facilities, each of which can be implemented within the features and mechanisms of its predecessor. We must begin with the primary limitations of existing facilities: their lack of a conceptually elegant data structure and their lack of a conceptually elegant modularization mechanism.

A functional language supporting generalized arrays reduces these limitations. The functional notation provides the modularization mechanism. The generalized arrays provide the data structure. The goal of this thesis is the development of such a functional notation, the development of the generalized-array data-structure, and the development of a suitable system-conceptualization for the support of these features in a work-bench environment.

The resulting work-bench can then be used by itself, for solving problems that are difficult to solve with existing work-bench facilities. It should be particularly useful to those problem-solvers who find the existing work-bench facilities too simple-minded for their abilities. Furthermore, the results of this experiment can be used as a level in the bootstrap hierarchy of more and more sophisticated programming facilities.

Preface

> In this thesis, direct quotation will be indicated by the left and right indenting of single spaced text.

The following typographical conventions will be used throughout this thesis.

> The first occurrence of a word in the sentence that defines it will be set in **bold** type.
>
> An ordinary variable will be set in lower case *italic* type.
>
> A variable restricted to the domain of generalized arrays will be set in upper case *APL* type.
>
> A specific generalized array will be set in upper case **BOLD** type.
>
> An arbitrary function will be represented by the delta symbol (∆).
>
> A specific function introduced for discussion will be set in lower case **bold** type.

The format used for proofs is

$$\begin{array}{rll} \text{antecedent} \leftrightarrow & \text{consequent} & \text{rule} \\ \leftrightarrow & \text{consequent} & \text{rule} \\ \leftrightarrow & \ldots & \ldots \\ \leftrightarrow & \text{consequent} & \text{rule} \end{array}$$

In the first chapter, and in the first three sections of the second chapter, the notation will be informal, and will be chosen to suit the discussion. From section 2.4 on, the formal notation of section 2.4 will be followed.

Table of Contents

List of Figures

Chapter One: Foundations

> *"What we need is imagination. We have to find a new view of the world."*
>
> Richard P. Feynman

The arguments presented in this thesis are intended to support the hypothesis that the generalized array is the arrangement best suited for the conceptualization of data structures. Arguments will be presented in favor of the use of generalized arrays, embeded in a functional notation, for the discussion of system structure and for the conceptualization of algorithms.

This first chapter reviews both the limitations experienced by humans solving complex problems, and the modularization of solutions to permit their comprehension. The hypothesis is presented and related to the complexity limitations. The concepts discussed in the remaining chapters are introduced.

1.1: Psychological Aspects of Complexity

- *Redundancy Allows Simplification.*

The central task of a natural science is considered by Herbert Simon[1] to be the detection of patterns hidden in the environment being studied. He finds[2] that it is the redundancy within a complex structure which allows the complex structure to be described by a simple structure. A structure with no redundancy can only be exhibited, since any simplification would result in a loss of information.

As a simple example, consider the summation of the integers from one to *n*. A person could be instructed to perform this feat with the command "add one to two and to the result add three and to the result add ... and to the result add *n*-1 and to the result add *n*." The redundancy in this solution can easily be detected in the recurring phrase "and to the result add." An encoded formulation would make use of the simpler command "begin with one and then add each number from two through *n* to the sum obtained so far." This is a simplification of the description, at the cost of the increased interpretation performed by the processer. Were

[1] Simon, Herbert A., *The Sciences of the Artificial* (Cambridge, Mass.: The M.I.T. Press, 1969), p. 1.

[2] Ibid., p. 110.

the processer able to perform multiplication and division, and interpret algebraic expressions, the entire process could be encoded as $n \times (n+1) \div 2$. But if no such redundancy can be found, or if no processor can be found that understands the encoding, then the complete description must be provided.

It is exactly this pattern-description task that is central to the design of algorithms. The goal of algorithm design is a simple representation for a complex process, a representation encoding the redundancy of the process as an algorithm. For the designer, the representation *is* the solution.

> How complex or simple a structure is depends critically upon the way in which we describe it. Most of the complex structures found in the world are enormously redundant, and we can use this redundancy to simplify their description. But to use it, to achieve the simplification, we must find the right representation[1]

- *Human Comprehension Limitations.*

The methods that a problem-solver uses to discover a redundancy encoding are poorly understood[2] Nevertheless, once an encoding has been postulated, the hypothesis must be

[1] Ibid., p. 117.

[2] Wilson, Kellogg V., personal communication, December, 1979.

tested. Normally, this testing requires that the encoding be expressed by some abstract representation. The problem-solver must be able to comprehend the representation as an expression of the encoding.

Our short term memory is capable of retaining only about five chunks of information, and once we are interrupted, only about two chunks can reliably be remembered. Furthermore, there is a measureable chunk-transfer time of about five seconds between short-term and long-term memory![1]

Once the short-term memory capacity of a designer is exceeded, his ability to correctly test his hypothesised redundancy pattern deteriorates. Certainly, an experienced programmer is able to include far more information in a chunk than a beginner, just as an experienced chess player is able to treat an entire board situation or play strategy as a single chunk?[2] Even so, if a situation is too complex, considering the experience of the designer, then the following problems are encountered.

[1] The exact bound on the size of an information chunk is not known, but it is small. For a discussion of these limits, and of the memory transfer time, see Simon, *The Sciences of the Artificial*, p. 39-42., and Miller, George A., "The Magical Number Seven, Plus or Minus Two: Some limits on Our Capacity for Processing Information," *The Psychological Review 63*, 2(March 1956):81-96.

[2] Hofstadter, D. R., *Gödel, Escher, Bach: an Eternal Golden Braid*, (New York: Basic Books Inc., 1979), p. 285-287.

> First, the span of absolute judgment, and the span of immediate memory, impose severe limitations on the amount of information that we are able to receive, process, and remember![1]

> Subjects get into trouble simply because they forget where they are, what assignments they have made previously, and what assumptions are implicit in the assignments they have made conditionally?[2]

> In all the living systems we studied, as the information input rate went up, the output rate increased to a maximum and thereafter decreased, showing signs of overload[3]

1.2: Hierarchical Modularization

The solution to the memory problems discussed in the previous section is the hierarchical modularization of the problem being solved.

> In the best of all possible worlds—at least for the designer—we might hope to be able to characterize the main properties of the system and its behaviour without elaborating the detail of either the outer or the inner environments. We might look towards a science of the artificial that would depend on the relative simplicity of the interface as its primary source of generality[4]

[1] Miller, "The Magical Number Seven," p. 96.

[2] Simon, *The Sciences of the Artificial*, p. 35.

[3] Miller, James G., *Living Systems*, (New York: McGraw-Hill Book Company, 1978), p. 195.

[4] Simon, *The Sciences of the Artificial*, p. 9.

Whenever the solution to a problem is needed, and the details of the solution are hidden behind an interface, the complexity of the comprehension of the solution is reduced to the complexity of the comprehension of the interface. If the interface is simple enough, then its comprehension may only require a single chunk of short-term memory. If a particularly complex algorithm can be modularized in such a way that the individual modules perform simple tasks, and can therefore be easily understood, and if the interfaces between the modules are simple enough, then the entire hierarchy will be easier to understand than a monolithic solution, because the designer will be able to restrict his attention to one simple module at a time, in which each interface requires only a single chunk of short-term memory.

- *Tempus fidgets, Hora scopes.*

With the following parable, Simon illustrates the advantages of the hierarchical approach[1] Two watchmakers, Tempus and Hora, manufacture fine watches of about one thousand parts each. The watches are constructed by Tempus in such a way that any interruption, such as the answering of the telephone, causes the entire watch to fall to pieces. Conversely, the watches made by Hora are built of

[1] Ibid., p. 91-93.

hierarchical sub-assemblies, each sub-assembly consisting of about ten smaller sub-assemblies. Consequently, Hora loses no more than ten operations when a sub-assembly falls to pieces due to an interruption.

In Simon's analysis of the watchmakers, the expected time for the completion of a watch by Tempus can be computed from the expected time for the completion of a watch by Hora as

$$t_{\text{Tempus}} \simeq \frac{2}{n\,p} \times (1 - p)^{m-n} \times t_{\text{Hora}}$$

where

p is the probability of being interrupted while adding a part,

n is the number of elemental parts in the entire assembly, and

m is the number of sub-modules in a module.

In Simon's example, where n=1000 and m=10, if p=0.01 then Tempus would take about four thousand times as long as Hora to complete a watch. Figure 1.2-1 shows some of the effects of these parameters on the completion time ratio, for n=1000.

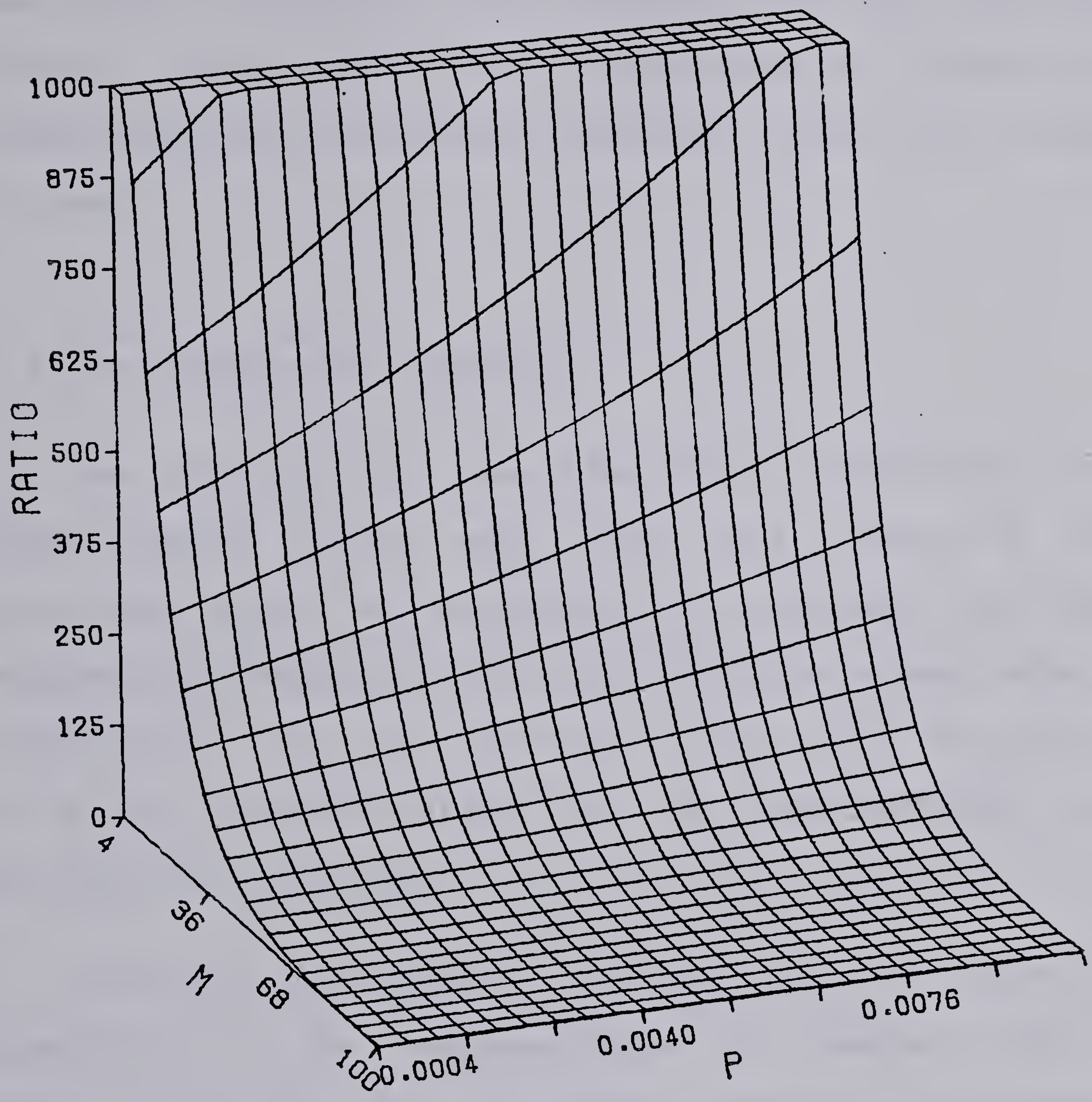


Figure 1.2-1: The Watch Completion Time Ratio

This parable accurately illustrates the monolithic software development problem. In the watchmaker metaphor, watches are equivalent to systems, sub-assemblies are equivalent to modules, and elemental parts are equivalent to chunks. The difficulties experienced by Tempus will be experienced by the software developer unless Hora's approach is used.

1.3: The Complexity Hypothesis

The previous two sections have established that the human problem solver must limit the complexity of the solutions which he attempts to comprehend, and that the hierarchical modularization of a solution accomplishes this limitation. In this section, our goal will be established to be the simplification of the *representation* of the solution to a problem.

Classical programming languages are not sufficiently powerful for the representation of complex algorithms. These languages are so closely coupled to the underlying machine, a particularly simple-minded machine, that they require tremendous detail even for the description of simple algorithms. Software teams attempting the development of ultra-large systems are beginning to realize that they are incapable of comprehending algorithms expressed in so complex a form.

The description of a solution is less complex when the intended processor has the capability to comprehend more complex operations without running out of memory chunks. Therefore, the complexity of the representation will be reduced by increasing the capability of the processor, in the same way as the winning ability of the chess player depends on his capability to chunk advanced chess concepts.

- *The State of Software Development.*

Terry Winograd[1] makes the following observations about the state of software development:

> [1] Computers are not primarily used for solving well structured mathematical problems or data processing, but instead are components of complex systems.
>
> [2] The building blocks out of which systems are built are not at the level of programming language constructs. They are "subsystems" or "packages," each of which is an integrated collection of data structures, programs, and protocols.
>
> [3] The main activity of programming is not the origination of new independent programs, but the integration, modification, and explanation of existing ones.

Winograd also makes observations about requirements that a representation must satisfy. In particular:

[1] Winograd, Terry, "Beyond Programming Languages," *Communciations of the ACM 22*, 7(July 1979):391-401

> *We need a consistent way of talking about modularization* and interaction between semi-independent modules which can be applied to system structure at all levels of detail.
>
> In many cases, much of what is now thought of as control structure can be implicit in the data structure, leading to notions of "nonprocedural" or "procedureless" languages. *The interaction between control and data structure needs to be put into a theoretical framework.*

Winograd goes on to consider many of the very high level features of existing application-oriented languages, in order to define the requirements of a very high level general purpose language. However, it soon becomes apparent that the development of such a very high level language would grossly exceed the very complexity limitations that already confound attempts at large systems, leading him to conclude that:

> There will have to be a very careful program of bootstrapping to get from today's languages and systems to the one I have described. The reason for writing a paper of this sort ... is the recognition that the relevant ideas need more development, and the hope that people will turn their attention to them.

- *The Dimensions of Representation Complexity.*

We have determined that we require a processor capable of comprehending more complex representations. We have determined that we will have to reduce complexity in those areas that are representative of the current trends in software development, and we have determined that we will

have to teach comprehension to our processors by adding small increments to their domain of comprehension. In what direction should the first increments be added?

> The "software crisis" is the result of our human limitations in dealing with complexity. To "solve" the problem we must reduce the "apparent complexity" of programs, and *this reduction must occur in the program text* .... We know something about the way humans have traditionally dealt with understanding complex problems ... and we can try to mold the expression of a program so that it facilitates these techniques[1]

The first increments to be added to the domain of comprehension of our processors must be directed towards the reduction of the apparent complexity of the program source code. The apparent complexity of the program source code is dependent on the number of chunks of information that the reader of the representation must attempt to manage. Therefore, the reduction of this apparent complexity depends on the reduction of the number of chunks of information that must be managed. How can we reduce the number of chunks of information that must be managed?

A formulation of this reduction task has been made by

[1] Wulf, W. A., "Some Thoughts on the Next Generation of Programming Languages," in *Perspectives of Computer Science,* ed. A. K. Jones, (New York: Academic Press, 1977).

Jacob Schwartz[1] He considers a program to be a set of elements. The comprehension of each element requires, in addition to the comprehension of the element itself, the comprehension of the entire set of constraints placed on the element by the other elements of the program. This set of constraints is known as the **local complexity** of the element. Psychologically, the probability of successfully understanding an element within its environment decreases rapidly as the local complexity of the element increases.

Schwartz points out that the inverse of this probability of success is a metric of the difficulty of understanding the program, and therefore is a metric of the time required to understand the program. Furthermore, he suggests that each programmer will have a local-complexity threshold, above which he will be unable to understand a program in a reasonable amount of time.

The reduction of the number of chunks that must be managed is therefore dependent on the reduction of three seperate complexities:

1 ◦ the reduction of the number of elements in the program,

2 ◦ the reduction of the complexity of the individual

[1] Schwartz, Jacob T., *On Programming: An Interm Report on the SETL Project*, (New York: Courant Institute of Mathematical Science, revised 1975), p. 1-3.

elements of the program, and

3 ∘ the reduction of the number of constraints that the elements of a program place on each other.

- *The Complexity Reduction Campaigns.*

Our approach to the reduction of these complexities can be conveniently divided into the following three campaigns.

The first approach, known as the **intra-module** campaign, will attack the complexity of the expressions within the modules. Its goals will include the reduction of the requirements for sequencing information that is not inherent in the problem, the reduction of the requirements for the specification of algorithms to manage data structures, and the reduction of the requirements for the specification of detail. These reductions will be achieved by extending the knowledge of the processor until it can infer the sequences, data structure management, and details of the solution from the context of the less complex description. As a result of the intra-module campaign, *the responsibility for many of the constraints in the local complexity of the elements will be transferred from the programmer to the machine.*

The second approach, known as the **modularization** campaign, will attack the local complexity of the elements of a program by moving elements out of the program. Whenever a subset of the elements of a program can be

transferred to a well-defined module with well-defined imports and well-defined exports, *the local complexity of the elements will be decreased, since the programmer will no longer have to comprehend the constraints originally imposed by the elements of the removed subset.* (Of course, the local complexity of the removed subset will be similarly reduced.)

The third approach, known as the **trans-module** campaign, is in fact required to support the modularization campaign. When modules are dependent on the internal workings of each other, a change that is required in one module may require some other module to be changed in order that the two remain in concert. However, the subtlety of the interdependance may hide the consequent change until the system with the antecedent change is tested, consequently, a second release of the system may be necessary. Similarly, the changes made in the *n*'th release may require changes to be made in the *n*+1'th release.

Consider a matrix in which each *ij*'th element is the probability that a change required in module *i* will require a change in module *j*. Haney[1]

[1] Haney, Frederick M., "Module Connection Analysis—A Tool for Scheduling Software Debugging Activities," in the Fall Joint Computer Conference, (Montvale, N. J.: AFIPS Press, 1972) 41:173-179.

shows that when the principal eigenvalue of the interconnection matrix is greater than one, the system of modules is so unstable that positive feedback may cause a single required change to generate an ever-increasing number of errors. Figure 1.3-1 shows a typical relationship between the average interconnection probability, the number of changes required, and the number of times required changes have been made. As a result of the trans-module campaign, *the stability of the system will be guaranteed by rigorously enforcing the interfaces between the modules.*

Overall then, this is a proposal for a language experiment. It is an experiment in the reduction of the complexity of the representation of algorithms. It is an experiment intended to move programming one step away from the hardware's monolithic, paleolinguistic software: one step towards the problem-solver's requirements.

1.4: The Langugage Classifications

The intent of this section is the classification of both the negative characteristics of the classical languages and the alternatives that can overcome their disadvantages. The distinctions between the procedural and the functional

[1] (cont'd)

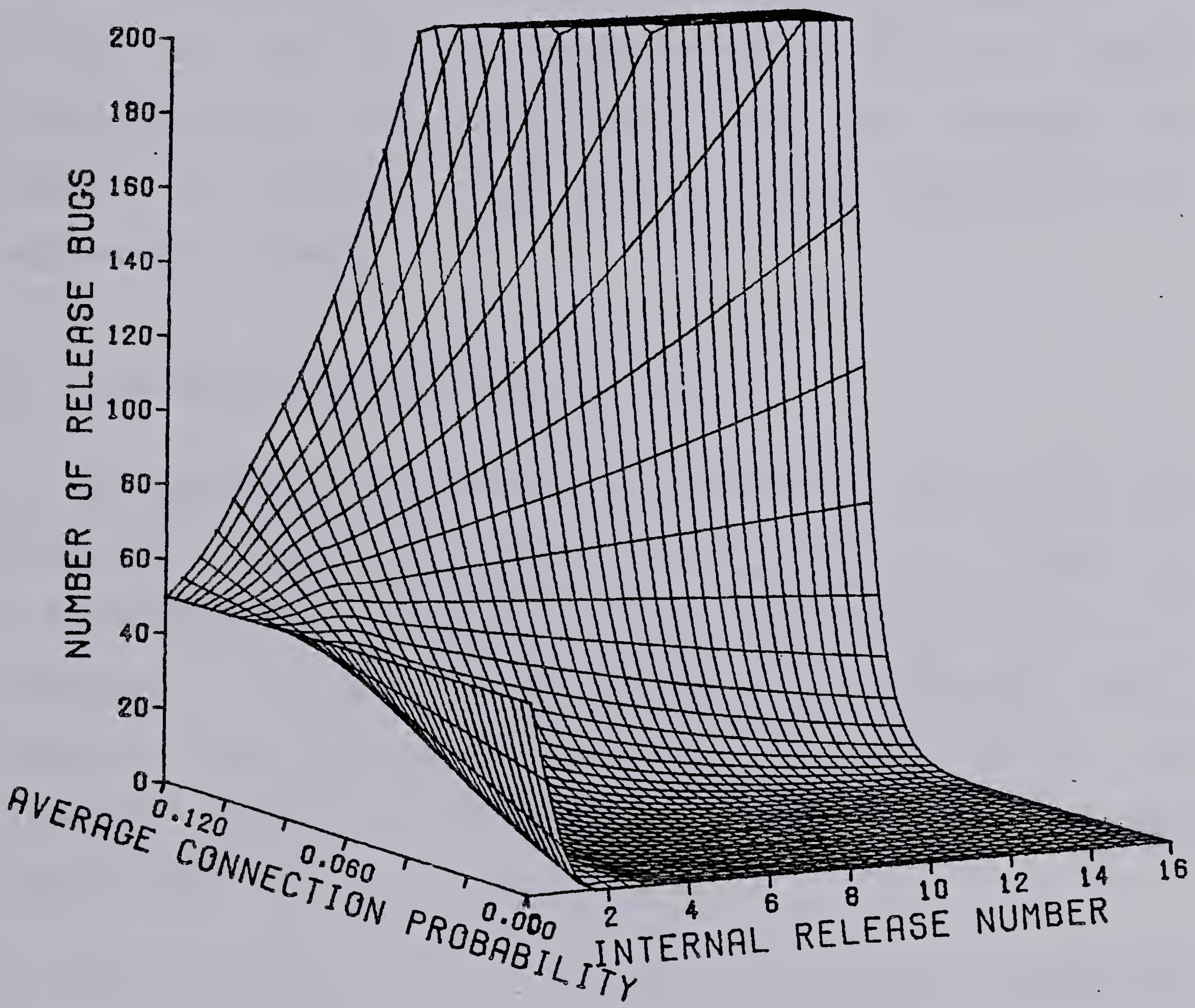


Figure 1.3-1: The Number of Changes Required

language classes are defined. The distinctions between piece-wise data-structures and aggregate data-structures are made. These two distinctions are seen to define a plane on which a language may be located relative to the location of the other languages on the plane. A particular language will rarely be purely functional or purely procedural, or allow only the piece-wise manipulation of data or support fully aggregate data-structures. But two languages can usually be ordered with respect to their functionality or aggregation power.

- *The Sequencing-Information Axis.*

The distinction between the procedural and functional language classes depends on the amount of sequencing information that must be specified by representations in the language. As explained below, procedural languages tend to require the addition of sequencing specification not inherent to the problem being solved. Functional languages tend to minimize this requirement.

The solution to a problem may contain sequencing restrictions that must be represented in an encoding of the solution. For example, if instructed to move from the inside of a room to the outside of the room, through a door which is now closed but can be opened, then the solution will usually require the opening of the door *before* traversing the threshold, rather than after (thereby

allowing one to save face). In this case, the representation of the solution must encode the before/after sequence.

A **procedural** language is characterized by the abundance of sequencing information that is incidental to the solution being encoded in the algorithm, but is essential to the representation itself. For example, when a solution requires a function to be applied to each member of a subset of a data structure, and the solution does not require the application to be carried out in any particular order, then the management of a particular order just increases the local complexity of the other elements in the representation of the solution.

Typically, a procedural representation is a description of *how* to effect the solution, as opposed to a description of *what* the solution is. Although this distinction is not often intuitivly elegant, that is only because almost all of the languages that exist today are procedural languages! The examples discussed later should provide insight into the difference.

[1] Leavenworth, B. M., and Sammet, J. E., "An Overview of Nonprocedural Languages," in Proceedings of a Symposium on Very High Level Languages, *ACM SIGPLAN Notices 9*, 4(April 1974):1-12.

A **functional** language is characterized by a set of requirements. Functional languages have no statements. Each function in the language is just an expression. There is no assignment in a functional language and consequently there are no variables, and so no existing information is ever altered in the evaluation of a function. There are no parameters in a functional language, the functions are simply applied to their argument.

Actually, these requirements are unimportant. The essential characteristic of a functional language is that it tends to minimize the requirements for the addition of sequencing information to the solution, and it tends to specify what a solution is, instead of how to carry out the solution.

- *The Data Aggregation Axis.*

The classical programming languages are based on the classical computer; consequently they share many of the limitations of these devices. In particular, most of them share the very low level on which data items are manipulated. Much of the complexity of classical languages can be traced to this low-level view, and much of it can be eliminated by switching to a high-level view.

A classical computer is characterized by a central processing unit, with a small amount of internal storage, connected to an external storage device via a narrow communication channel. All of the computer's instructions and data are stored in the external medium and must be transported to the processor via the communication channel. In order to accomplish this transfer, the location of the instruction or data item must be obtained, and unless this location can be computed from the information already within the small amount of internal storage, more information will have to be obtained from the external medium. Of course, this often leads to the problem of finding the location of the location, and so on. Typically, this results in a bidirectional traffic bottleneck at the communication channel.[1]

As a result of the channel-bottleneck constraint, programming has largely become an exercise in controlling the traffic at the bottleneck, rather than a study of the overall behaviour of the programs. Classical languages are characterized by their assignment-based operations, each imitating the single-word fetch and store instructions of

[1] Backus, J. W., "Can Programming be Liberated from Its von Neumann Style? A Functional Style and Its Algebra of Programs," 1978 ACM Turing Award Lecture, *Research Report RJ2234(30357)4/25/78*, (San Jose: IBM Research Laboratory, 1978), p. 6-10.

the machine and, by their control structures, each imitating the branch instructions of the machine.

Certain more powerful languages are characterized by their aggregate data structures. In *APL*, for example, every data item is a rectangular array of arbitrary shape, and in LISP, every data item is a binary list of arbitrary nesting. These languages are characterized by their data structures because their primitives are designed to use them. In the classical languages the primitives are designed to handle only very limited data structures, such as linear strings and simple scalars[1]

The advantage of these aggregate data-structures is that a solution representation which makes use of these aggregate data-structures does not have to supply information on how to use the aggregrate data-structures. Since this information does not have to be described, the complexity of the representation is reduced. In order to minimize the amount of information that an algorithm must supply about its data structures, we will use a generalized data structure that mirrors as many of the properties of real data structures as possible.

[1] A data structure built of pointers is just a vector of scalars with an interpretation supplied by the software. It is not an aggregate data structure.

A **generalized array** is such an aggregate data structure. Athough the theory of generalized arrays considered herein does not define an array (in the same sense that set theory does not define a set), an intuitive notion of a generalized array can be given. Generalized arrays have no shape or nesting constraints. Every element of a generalized array is a generalized array. A generalized array is rectangular. Any slice normal to any particular axis will be of the same shape as any other slice normal to that same axis. A generalized array is of arbitrary dimension, that is, it can have any number of axes.

- *The Language Classification Plane.*

As has already been noted, most languages cannot be absolutely classified[1] into any of these categories. However, some languages can cleary be seen to be either more or less functional than other languages, and certain languages can be seen to handle either more or less aggregate data-structures than other languages. Together, these two axes of differentiation define a plane on which a language may be located.

[1] Leavenworth, B. M., and Sammet, J. E., "An Overview of Nonprocedural Languages," p. 3.

In review, one axis distinguishes between functional languages and procedural languages, and one axis distinguishes between piece-wise data-structures and aggregate data-structures, where

1 ∘ procedural languages tend to specify the steps involved in the execution of a solution, and consequently tend to require the addition of sequencing information to the representaton of the solution,

2 ∘ functional languages tend to specify what the solution is and therefore tend not to require the addition of extra sequencing information,

3 ∘ piece-wise data structures tend to be closely coupled to the classical computer and therefore tend to require the addition of extra information for the interpretation of data structures, and the addition of the associated extra sequencing information and local complexity, and

4 ∘ aggregate data-structures tend to model the properties of real data-structures and therefore tend not to increase the complexity of the representation.

Since both the procedural characteristics and the piece-wise data-structures are related to the classical computer, most languages tend to be burdened with both of them. On the other hand, since both the functional characteristics and the aggregate data-structures tend to

reduce the complexity of the solutions that make use of them, the language of this experiment will be functional, and will handle generalized arrays.

Chapter Two: A Notational Framework

> *"The formalization of a science usually admits of the possibility of introducing new signs into that science which were not explicitly given at the outset."*
>
> Alfred Tarsky

A functional notational-framework is developed in this chapter. In the first section, the classical notion of control structures is examined, and the control structure requirements of the functional notation are determined. The second section examines the concept of history, and determines the history requirements of the functional notation. The third section defines the computational model of the system that will be attempting to understand the solutions that we represent to it in our functional notation.

Section four marks the beginning of the formalization of this language experiment. In this section, the syntax of the functional notation is developed. Section five then expands the syntax to provide an easier method for the use of meta-functions.

The goal of this language experiment has been determined to be the examination of the generalized array data structure and some of the other alternatives to the

classical programming languages. The definition of a particular alternative language is not a goal. Consequently, the notational and computational framework developed here is fairly arbitrary, and need only be representative of the concepts of the experiment and convenient for the experiment itself. Furthermore, there is no need for complete rigor in the development of this framework, and so no attempt will be made to be completely rigorous.

2.1: The Control Structure Alternatives

For the purposes of this discussion, a **control structure** is considered to be a method for the alteration of what would otherwise be a linear sequence of representaion. Procedural languages have only two classes of control structures: a repetition class (typified by the *while* structure) which causes a set of instructions to be repeated, and an alternative-selection class (typified by the *if* structure) which selects one of a set of alternate paths to follow through the instructions. This section reviews the reasons why these strutures exist, and determines the control structure requirements of our functional notation.

- *The Alternative-Selection Control Structure.*

All alternative-selection control structures are based on the conditional construct. A **conditional** construct is one which selects between two alternate sub-sections of a representation depending on the value of an expression. The conditional construct is usually used in one of two applications.

Firstly, the conditional construct is used when a selection must be made between a section of the representation to be used when the solution is being considered at the boundry of its applicability, and a section to be used when not at that boundry. For example, in the factorial program

f(n) IS IF n = 0 THEN 1 ELSE n×f(n-1)

the conditional construct selects between the expression to be used at the boundry n=0 and the expression to be used when not at that boundry.

Secondly, the conditional construct is used when a selection must be made between two sections of a representation that are applicable at different boundries, for example

g(m,n) IS IF m = "yes" THEN **p**(n) ELSE **q**(n).

where the value of m is either "yes" or "no". In this

example the value of g(m) at the boundry m="yes" is p(n) and is q(n) at the boundry m="no".

The theory of generalized arrays which will be presented in Chapter Three obviates much of the need for the alternative-selection concept.

> Even though the if-then-else construction is certainly useful and necessary in many areas of programming, the power of this conditional to circumvent any difficulty represents an inherent weakness in programming as a conceptual discipline![1]

Instead of relying on the alternative-selection construct, an axiomatic theory which does not require such a construct can be developed. For example, if the system processing the factorial function f already knows the axiom f(0)=1, then the definition f(n) IS n×f(n-1) is sufficient, because the system will not invoke this defintion when n=0, but will simply use the axiom. The discussion of generalized arrays in Chapter Three will make use of this axiomatic alternative, and will not use the conditional construct.

Even though this axiomatic alternative to the conditional construct works well when discussing the primitives of our language, it fails when we discuss more

[1] More, T., "The Nested Rectangular Array as a Model of Data," *ACM-STAPL/SIGPLAN Proceedings APL79 Conference*, (May 1979):69.

involved applications of the primitives. The function g illustrates a typical situation in which the boundry conditions may not be able to be incorporated in the process as axioms. In these *applications*, the conditional structure will be needed.

Therefore, the complexity costs of the following special structure can be justified! The expression

$$P1 \rightarrow Q1;\ P2 \rightarrow Q2;\ \ldots;\ Pn \rightarrow Qn;\ D$$

is read as: if $P1$ then $Q1$ else if $P2$ then $Q2$ else if ... else if Pn then Qn else D. The value of the expression is the value of the subexpression Qi which corresponds to the first subexpression Pi which evaluates to a logically true value. If no Qi is true then the result is D. In pure LISP this service is provided by the COND function. A method for implementing this structure entirly within the formal notation of this chapter will be presented in Chapter Four.

This conditional form is familiar from the predicate calculus expression $(P \rightarrow Q) \wedge (\sim P \rightarrow R)$. Another level of nesting can be handled with an expression of the form

$$((P \wedge Q) \rightarrow R) \wedge (P \wedge \sim Q \rightarrow S)) \wedge (\sim P \rightarrow T).$$

[1] McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1," *Communciations of the ACM 3*, 4(April 1960):184-195.

In order to prevent an increase in the local complexity of the elements of a representation, further levels of nesting should be accomplished either with the introduction of additional modules, or with the implementation of a table-driven representation. This enforcement of a well organized modularization of the solution into non-procedural functions is necessary to meet the objectives of section 1.2.

- *The Repetition Control Structure.*

The repetition control structure is used in exactly two cases: for the application of a section of a representation to each of a set of elements, and for the determination of the effect of a section which depends on the effect of the same section of the representation.

A procedural language uses its repetition structure whenever an algorithm is to be applied to a subset of the elements of an aggregate data structure[1] The most common form of this repetition is

$$\text{FOR } i \text{ FROM } 1 \text{ TO } n \text{ DO } x(i) = \mathrm{f}(y(i)).$$

A functional representation of this repetition is typically

[1] More, T., "The Nested Rectangular Array as a Model of Data," p. 61-63.

of the form

APPLY f TO EACH y

or f:y, which does not require the temporary counter i, does not require explicit knowledge of n, the shape of y, and does not require the temporary variable x, in which to store the result.

The common axis operations (such as *APL*'s +/[1]*X*) can typically be described in a functional form as

APPLY f TO AXIS n OF x

or f*n:x, but requires the complex procedural form

```
z = a
FOR i FROM 1 TO n DO
    FOR j FROM 1 TO m DO
        z(i) = f(z(i),x(i,j));.
```

These axis operations are a variant of the subset-application case of the procedural repetition-structures. The procedural form is vastly more complex. It requires the maintenance of three temporary variables, the introduction of two sequence orders not required in the functional form, explicit knowledge of the shape of x, and initialization of the result. The application of f to a different axis requires alteration of the subexpressions of the procedural form, but requires only a value change in the functional form. The syntax of the procedural form depends on the number of axes, whereas the

functional syntax is invariant.

A procedural language also uses its repetition control structure whenever the value of a function depends on a previous value of the function.[1] A common example of this form of repetition is the factorial program

$$n! \text{ IS } f = 1;\ \text{FOR } i \text{ FROM } 1 \text{ TO } n \text{ DO } f = f \times i;\ f.$$

A functional form of this program would take advantage of the boundary condition (0! IS $f = 1$; 1) and the recurrence relation $i! = i \times (i-1)!$, to define the recursive form

$$n! \ \ IS\ (n=0) \rightarrow 1;\ n \times (n-1)!.$$

The functional form prevents the representation from having to maintain the two temporary variables f and i.

Even this recursive solution can introduce more sequencing information than is required. The recursive factorial algorithm can be replaced by the even more functional form:

$$n! \text{ IS the product of the first } n \text{ integers.}$$

This second form is similar to *APL*'s $\times/\iota N$. Notice that this more functional form also relies on the axiomatic behaviour of the system at the boundry to prevent the use of the

[1] Burge, "Recursive Programming Techniques," p. 18-21.

conditional construct.

Many other linear sequence problems, such as reading a book, are naturally solved with recursive algorithms like

```
                    read-rest(book)
                          IS
      read-page first(book), read-rest rest(book),
```

and are easily programmed in such a form.

- *The Control Structure Requirements*

These control structure classes are an overwhelming complexity burden for the programmer. If a problem explicitly requires the type of control provided by a recursive solution or explicitly requires the selection of unrelated alternatives (that cannot be generated without selection) then the recursive or alternative-selection functional forms can be used. Otherwise, the responsibility for the maintenance of the control structures that specify *how* to obtain the solution should be transferred from the programmer to the machine, and the programmer should return to the consideration of *what* is required of the solution. A functional language allows this transfer of responsibility.

2.2: The History Sensitivity of Functional Languages

Data that must be stored for later retrival by a name that is associated with it at the time of storage is **historic** information. Since a purely functional language has no statements, no assignment, and no variables, a purely functional language has no method for storing values for later use. For the purposes of the study of the theory of generalized arrays, this will present no problem, and will actually allow for the very elegant expression of the theory. In fact, it does not ever present any theoretical problem because the entire environment can, in principle, be passed into the top-level function and the necessary parts of that environment can be selected wherever required.

This history-independence decreases the apparent complexity of representations by eliminating the management of historic information from the representation. In the functional form of an algorithm, values are calculated where they are needed. Since there are no variables there is no need to worry about program side-effects, order-of-execution problems, or name-scope control![1]

[1] Wiedman, Clark, "*APL* Problems with Order of Execution," University Computing Center, Graduate Research Center, University of Massachusetts, Amherst, Mass.

On the other hand, the expensive re-calculation of values may seriously impact the performance of a program. A program's inability to access values that were previously generated and its inability to save values for later use represents a major restriction. The decrease in history complexity may actually offset the recalculation costs; however, historic information generated in some other environment may occasionally be required, or a program may wish to generate historic information and later re-access it in order to prevent exorbitant recalculation costs.

In such cases, rather than destroy the functionality of the language, our system model will include a history process which runs in parallel with the other processes on the system. This history process will simply maintain a list of all the historic information sent to it via the communication facilities of the system, and will, on request, be able to send the information back to a process.

The use of this history process is similar to the use of the input/output operations of a procedural language. Its use is no more complex than the assignment operations of the procedural languages, but remains functional. Since it is a part of the support system, and not the language, it does not have to be axiomatized at this time (but might be once experience with its use has been gained).

A functional output attempt accepts as its argument a name for the data (somewhat like a file-name in a procedural WRITE) and the data itself. The data is remembered as being associated with the particular name. The result of the primitive is an indication of the success of the output attempt.

A functional input attempt accepts as its argument the name of a previously remembered item. The result of the primitive is either the item itself, or an indication of the failure of the input attempt.

2.3: A Computational Model

Even though our primary intent is the examination of the theory of generalized arrays, their actual usefulness will not be determined until their implications are interpreted in a number of application environments. In order to be able to interpret the implications of a set of axioms, we need:[1]

[1] Minsky, M., *Computation: Finite and Infinite Machines*, (London: Prentice-Hall International, Inc., 1972), p. 103-107.

1∘ a language in which to express the axioms and implications, and

2∘ a machine which can interpret statements in the language.

Of course, providing that an interpretive system is as powerful as a Universal Turing Machine, that interpretive system is capable of carrying out any effective procedure.

A statement-interpreting machine is just a string transformer, converting input strings (statements) into output strings (interpretations)[1] This machine model is the model of each of the set of processes in our system.

Even though our functional language is independent of any execution-environment considerations, it will be necessary to discuss certain such considerations. Of particular importance are the input/output operations introduced in the previous section and other operations for system communication such as the control functions for concurrent processes. Consequently, our functional framework must include a model of the framework in which the processes are interpreting.

[1] Post, Emil., "Formal Reductions of the General Combinatorial Decision Problem," *American Journal of Mathematics*, 65(1943):197-268.

The system model must be general enough to support the discussion of any real-machine consideration. At the same time, the model must be simple enough to prevent extraneous considerations from entering the discussion. The *APL* **shared-variable** model is one such model[1] A shared-variable system supports multiple concurrent processes and allows generalized arrays to be transferred between the processes.

With the shared-variable model any particular process sees a structurally equivalent model of the system (with a different set of other-processes, of course) as shown in figure 2.3-1. The expression to be evaluated by a process is specified when the process is conceived. The process communicates with the rest of the world via the shared variable primitives.

2.4: A Formal Expression Notation

Natural languages are too cumbersome, ambiguous, and irregular for the description of effective procedures[2] Instead, we will use a formal language. A formal language

[1] Lathwell, R. H., "System Formulation and *APL* Shared Variables," *IBM Journal of Research and Development 17*, 4(July 1970):356-357.

[2] Kleene, S. C., *Introduction to Metamathematics*, (Amsterdam: North-Holland Publishing Co., 1952), p. 59-62.

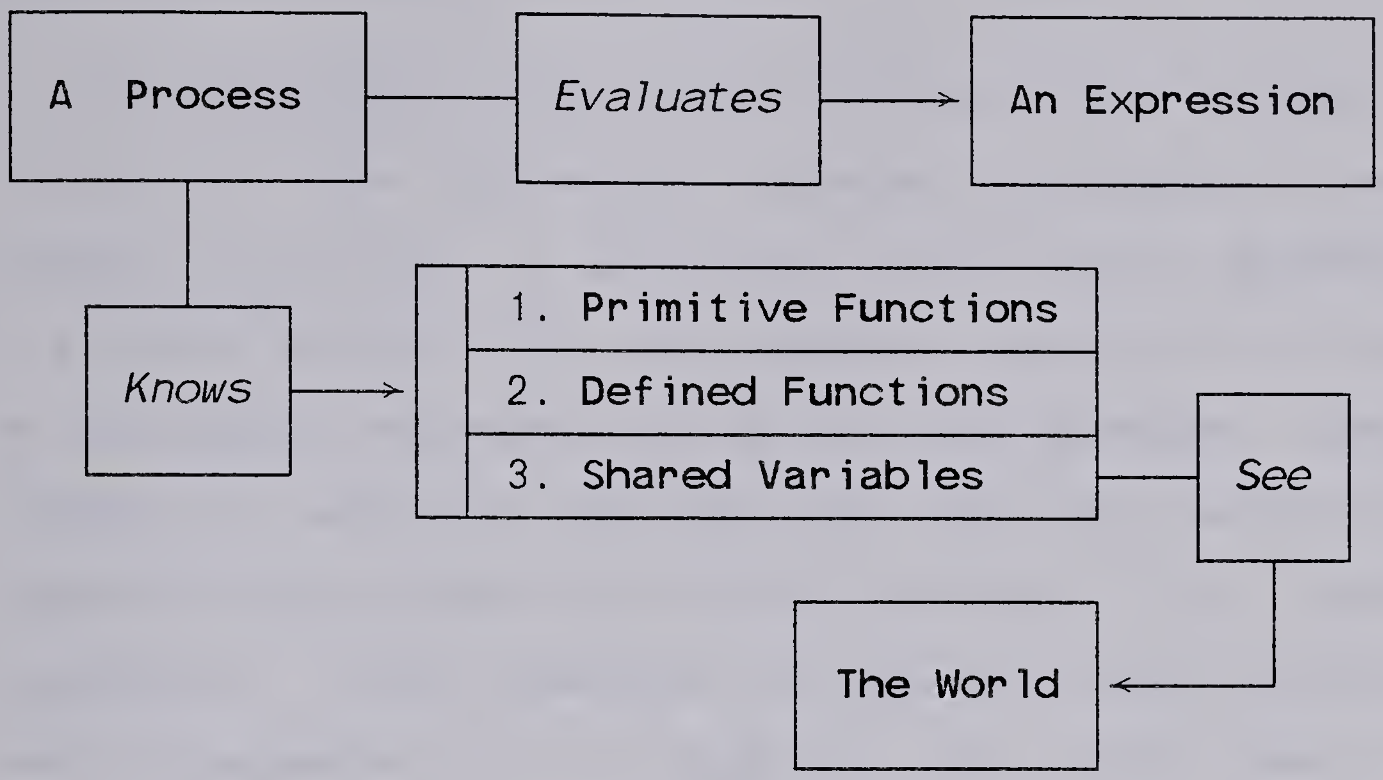


Figure 2.3-1: System Model As Seen By A Process.

is described by a set of formal symbols and a method for building formal expressions from the formal symbols[1]

- *The Formal Character Set.*

The processes of the previous section process strings of characters. Each expression of the language is such a string. For the purposes of this experiment, it is convenient to be able to use single characters to represent the primitive functions of the theory, and so the set of

[1] Ibid., p. 69-81.

characters to which we have access is enhanced by allowing simple symbol-combination.

The characters sent to and received from a process are related to the characters received by and sent from the process as follows. A user sees a collection of **graphics** on his output device. The three graphics ⍎⍀*B*⃞ may, for example, be the output representation of the five **character** sequence ⍎⍀*B*_bs_⎕ in which _bs_ denotes the character which causes adjacent output characters to be overstruck (the backspace character). That character sequence is composed from the set of **symbols** {⊥,∘,\,-,*B*,_bs_,⎕}, and could have been generated by the **keystroke** sequence ⊥ _os_ ∘ \ _os_ - *Q* _er_ *B* _bs_ ⎕ in which _os_ is the keystroke which causes symbols to be composed into characters (overstrike) and _er_ is the keystroke which erases the previous character from the string (error).

The process sees only the characters generated by the user's keystrokes and the user sees only the graphics generated by the characters sent to his output device. The error and overstrike keystrokes are never seen by the process, and the backspace symbol is never seen by the user.

Certain symbol and character subsets will be reserved for particular purposes, as an aid to the reader of the expressions. The symbols *A* through *Z*, all the characters composed of a symbol from *A* through *Z* and any other

character, and the underbar character (_), are the set of **alphabetic** characters. The **digits** 0 through 9, and all the characters composed of a digit and any other character form the set of **numeric** characters.

- *The Formal Symbols.*

The formal symbols of the language are built out of characters. An expression is broken into formal symbols as follows.

1 ∘ Any character which is neither alphabetic nor numeric is a formal symbol itself (except a dot (.) between two numeric characters).

2 ∘ Any substring consisting only of alphabetic characters and numeric characters, beginning with an alphabetic character, and delimited on both sides by a character which is neither alphabetic nor numeric, is a formal symbol.

3 ∘ Any substring consisting only of numeric characters, and dots, delimited on both sides by a character which is neither alphabetic, numeric, nor a dot, is a formal symbol.

4 ∘ Other than their use as formal-symbol delimiters, strings of blanks serve no formal purpose.

5 ∘ The parentheses ((and)), apostrophe ('), and dot (.) formal-symbols are reserved for use as grouping symbols.

6 ∘ Formal symbols consisting only of numeric characters and

dots represent numeric constants.

7 ∘ Strings of formal symbols containing only paired apostrophes, and delimited at each end by a single apostrophe, represent character constants.

8 ∘ Formal symbols which are neither grouping symbols, numeric constants, nor character constants, are the names of functions.

A numeric constant may or may not be well-formed. A well formed numeric constant consists of one or more digits, followed optionally by a decimal point followed by one or more digits. This syntax may be easily extended to include a formal notation for scientific, rational, and complex values, but for our discussion of array theory these numbers will be sufficient.

- *The Rules of Scope.*

Once an expression has been parsed into formal symbols, the meaning of the expression can be determined by applying each named function to the value of its arguments. The juxtaposition of function names will be assumed to imply the composition of the functions[1] The juxtaposition of two or more function names is ambiguous without rules of scope to

[1] Composition is an operator to be discussed in chapter three. When two functions are composed the result of one becomes the argument of the other.

determine the implied grouping of the composition arguments. The rules of scope are as follows:[1]

Every formal function which expects a right argument takes, as its right argument, the shortest well-formed expression to its right. If the function expects a left argument then the function takes, as its left argument, the longest well-formed expression to its left. These are known as the **short right scope** and **long left scope** rules. Numeric and character constants behave like functions which take no arguments and which have as their value the constant denoted by the name.

The parenthesis grouping-symbols alter the scope rules, allowing extended right scope and limited left scope. A parenthesised expression is well-formed only if all the left parentheses are balanced by corresponding right parentheses. A dot (.) suffixed to a function name expecting a right argument extends the right scope to the longest well-formed expression to the right. A prefix dot re-asserts the long left scope of a function name appearing in the extended right scope of some other function. There is no other implied hierarchy in the expressions.

[1] More, T., "Axioms and Theorems for a Theory of Arrays," *IBM Journal of Research and Development 17*, 2(March 1973):138-139.

tfu g. (v*hw* .j. x*ky*) l *z*

↔ *tfu* g. ((v*hw*) j. x*ky*) l *z*

↔ *tfu* g (((v*hw*) j (x*ky*)) l *z*)

↔ (*tfu*) g (((v*hw*) j (x*ky*)) l *z*)

where *t* through *z* expect no arguments, and f through l expect left and right arguments.

Figure 2.4-1: Function Scope Example

An expression can be converted to the fully parenthesised form as follows (see Figure 2.4-1 for an example). Replace each prefix dot (proceeding from left to right) with a right parenthesis, and a corresponding left parenthesis as far to the left as possible without altering the scope defined by the parenthesis already in place. Then replace each suffix dot (proceeding from left to right) with a left parenthesis, and a corresponding right parenthesis as far to the right as possible without altering the scope defined by the parenthesis already in place. The well-formed parenthesis rules and the long-left/short-right scope rules now completely determine the scope extent of all the names.

This set of scope rules has been chosen from a number of equally viable alternatives. Scope rules can provide for long or short scope for either argument of a function. Certain functions (the operators in *APL*, for example) could

have different scope rules than other functions. The expressions could be in prefix or postfix form instead of in infix form.

It has be argued, for example, that *APL's* expressions read constructively from left to right and analytically from right to left. On the other hand, experience shows that the readers of *APL* expressions parse them from both ends and within parenthesised subexpressions![1] The adoption of the mirror syntax simply reverses the directions for both cases.

The particular scope rules explained above are compatable with More's work. They have an unexpected advantage in that there need be no special symbol to indicate the sign of a numeric constant, because the short right scope allows the negate function to be applied to a positive constant to yield a negative constant, without having to introduce any additional grouping symbols.

- *The Denotation of Expressions.*

Formal systems usually include a naming concept.[2] Instead of having to exhibit an expression whenever it is to be referred to, a name can be associated with the

[1] Iverson, K. E., personal conversation, February 1980.

[2] Kleene, S. C., *Introduction to Metamathematics*, p. 70.

expression. The name then **denotes** the expression, and can be used instead of the expression in any place that the expression could itself be used. Such a named expression is known as a **function**.

The formal symbol $\equiv$ (read as **is**) is used in this experiment to indicate the association between a name and an expression. A name **n** may be associated with an expression *e* with the expression

$$\mathbf{n} \equiv e$$

which then allows the name **n** to be used in place of the expression *e*. Although the denotation is, for the purposes of the discussion of the array theory, considered to be a meta-operation, a name manager similar to the history process can easily be included as a process in the shared-variable model.

- *The Rules for Substitution*

In addition to a naming convention, formal systems usually include a substitution rule[1] For example, Church's lambda-calculus allows explicit substitutions to be

[1] Kleene, S. C., *Introduction to Metamathematics*, p. 78.

performed in an expression[1] On the other hand, the FFP notation of Backus avoids explicit substitution by distributing a construction of functions over an assumed left argument[2] In this experiment, substitution will be performed with the following **alpha-omega** substitution rule.

Whenever the name of an expression is interpreted, any occurrence of the formal symbol α will be replaced by the value of the left argument of the name, and any occurrence of the formal symbol ω will be replaced by the value of the right argument of the name, according to the scope rules.

When a function name is adjacent to an object on the left, then a definition of the name that contains an α must be used. When a function name is adjacent to an object on the right, then a definition of the name that contains an ω must be used. The **valency** of a function can be determined by the occurrences of α and ω in the function definition. A name definition with neither an α nor an ω is a **niladic** definition. A definition with only an α is **left-monadic**. A definition with only an ω is **right-monadic**. A definition with both α and ω is **dyadic**.

[1] Church, A., *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies, Number Six, (Princeton: Princeton University Press, 1941).

[2] Backus, J. W., "Can Programming Be Liberated...," p. 28.

2.5: Functional Forms

The language framework proposed so far has a serious drawback. There is no way to interpret the meaning of an expression encoded as an object of the language. The power of the Universal Turing Machine includes the ability to interpret the description of a machine encoded on the input tape. This ability to manipulate the descriptions of expressions is required here.

When the expressions of a language are made up of the same characters as the objects of the language, an object can be used to represent an expression. In order to prevent the evaluation of the expression, it can be enclosed in quotation symbols and thereby become an object itself. The object can then be manipulated by other expressions, and can be evaluated when desired by an evaluation function like execute (⍎) in *APL*, or EVAL in LISP. This evaluation function is the UTM, a process in our model. One can be explicitly invoked with the primitive function **interpret**, which is denoted by the formal symbol i-beam (⌶). The application of the interpret function to an expression encoded as an object yields the **meaning** of the object, that is, the expression itself.

Although this solution is theoretically complete, it is difficult to use because of the explicit quoting and interpretation required. If *APL* required this of its

operators then we would have to write something like ⍎'*A*',('+'.'×'),'*B*' instead of just *A*+.×*B*. As an alternative, the arguments of a function can be forced to be interpreted as functions whenever the function expects its arguments to be functions. All that we need to know is that the function expects a function as its argument and the substitution symbol will represent a function instead of an object.

A modified substitution symbol is used to represent a function argument. A left function-argument is indicated by the formal symbol ᾱ and a right function-argument is indicated by the formal symbol ω̄. Each occurrence of ᾱ will be replaced by the left-argument function and each occurrence of ω̄ will be replaced by the right-argument function. The entire expression, with the substutions, is then interpreted as the definition of the resulting function.

Any function which expects a function argument is a member of the special class of functions known as **operators**. Operators have up to two arguments (ᾱ and ω̄) and are function valued. Operators are also known as **transforms**.

The valence of a transform is determined in the same manner as the valence of a non-operator function except that the left-argument symbol is ᾱ and the right-argument symbol is ω̄.

Chapter Three: Generalized Arrays

> *"In times of general dispersion and separation, a great idea provides a focal point for the organization of recovery."*
>
> From the Huan Hexagram in the I Ching

As discussed in Chapter One, the complexity of solutions represented in the classical programming languages is greatly increased by their lack of aggregrate data-structures. Due to this lack of aggregrate data-structures, a representation has to include the explicit simulation of its aggregate data-structures using only the piece-wise data operations that are available. A sufficiently powerful aggregate data-structure removes this source of complexity, and allows algorithms to be conveniently expressed in a functional form. In this chapter the theory of the **generalized array** aggregrate data-structure is developed.

The first section of this chapter reviews the properties of real data-structures and relates these properties to the requirements of a general-purpose aggregate data-structure. The second section presents the axiomatic theory of generalized arrays. The third section extends the axiomatic theory to a practical version known as the applied theory. The fourth section reviews the results of the preceeding sections of the chapter.

3.1: The Selection of an Aggregate Data-Structure

The purpose of a generalized data-structure is the reduction of the complexity of the representation of a solution by allowing the easy representation of the data structures of the solution. Consequently, the generalized data-structure must be sufficiently powerful to easily represent any data structure that will occur in the representations.

- *The Properties of Actual Data.*

In order to determine the properties of a sufficiently powerful data structure, we will first examine the properties of actual data. The eight distinct properties that have been isolated by Trenchard More[1] will be presented here. Many of these principles have been previously discussed[2] but it is their joint effect that is important.

[1] A data structure should support the **Principle Of Aggregation**. A **collection** is said to **hold objects** as **items**. Items are said to **occur** at **locations** in the collection. Eggs, for example, often occur at depressions in crates.

[1] More, T., "The Nested Rectangular Array As A Model Of Data," p. 57-59.

[2] See, for example, Honig, W. L. and Carlson, C. R., "Toward An Understanding Of (Actual) Data Structures," *The Computer Journal 21*, 2(May 1978):98-104.

The crate holds the egg-objects as items.

[2] A data structure should support the **Principle Of Nesting**. Every item of an aggregation should again be an aggregation. In a library, for example, floors hold shelves, shelves hold books, books hold chapters, chapters hold sentences, sentences hold words and words hold letters.

[3] A data structure should support the **Principle Of Well-ordering**. All collections should have a **first** item. A well-ordered sub-collection is the result of the deletion of the first item of a collection. A new deck of cards, for example, is always in a specific order. Although unordered data structures are easily mappable into ordered ones by just ignoring the order, the converse is not true. It is much more difficult to manage the introduction of order into an unordered data structure, since it requires the management of the set-theoretic ordered pair.

[4] A data structure should support the **Principle Of Repetition**. Identical items occurring at different locations should be preserved. A box of animal crackers, for example, has identical animals occurring at different locations. If necessary, the identical items can be eliminated or ignored, but the retention of identical items at different locations in a data structure which does not support the principle of repetition requires the addition of unique identifying tags to the identical items.

[5] A data structure should support the **Principle of Valency**. The items of any collection should be organized along an intrinsic number of **axes**. This allows their access along different dimensions. A chessboard, for example, has two axes, the ranks and the files. Of course, if only one axis is required for a particular application, then such a structure may be used. On the other hand, simulation of a multi-axis structure with a linear one destroys the very convenient notion of adjacency. The moves of a king, for example, are defined in terms of traversal to adjacent squares. Furthermore, nested lists cannot distinguish between a two by three table of quadruples and a pair of three by four tables.

[6] A data structure should support the **Principle Of Smoothness**. Any section of items normal to a particular axis will have the same length. A typed page of paper can, for example, be considered to be a two-axis array of characters (with trailing space type, of course). On the other hand, if the page is to be considered to be a list of lines of variable length, then it can be so represented within these principles.

[7] A data structure should support the **Principle Of Arrangement**. By specifying the **position** of an item with respect to each of the axes, the item should be uniquely specified. In a city, for example, any corner should be uniquely specified by the names of the streets that

intersect there. The principle of arrangement does not, however, determine the structure of the address specification.

[8] A data structure should support the **Principle Of Orientation**. The specification of the location of an item should be a list of the locations of the item along each of the axes. This positional form will be chosen primarily for the purposes of closure, as discussed below. The principle of orientation determines the structure of the address specification.

If a single uniform data structure can support all of these principles, independent of their simulation by a representation, then that data structure will make the specification of real-data models much less complex. Otherwise, the representation will have to specify an instruction sequence and an encoding on a simple data structure, in order to represent the real data. This extra work increases the local complexity of the algorithms.

A linked list, for example, which makes a poor base for an indexed list, can be conveniently represented as a collection of two items in which the first item is the head of the list and the second item is the tail of the list.

A binary tree, for example, can be easily represented by a triple in which the first item is the value of the node, the second item is the left son, and the third item is the

right son.

A keyword data structure can be represented by a structure of pairs, in which the first item of a such a pair is the keyword and the second item is the value of the keyword. In the binary tree example, a triple of pairs would be used, the first items of which might be "value", "left-son", and "right-son" respectively.

Sometimes it is necessary for the data in a data structure to contain information about the data structure itself. A data structure containing such information is in fact a **meta-data-structure**. In many cases, instead of having a location hold an item, it may be desired to have the location hold the location of the item. This may be the case in a directed graph. Such a meta-data-structure is easily represented by having the location-holding locations actually hold the character string which describes the location. The target item may then be obtained simply by interpreting the location-holding item with the primitive interpret function. This sort of nesting, equivalent to the use of pointers in the classical languages, can be carried to arbitrary depth by including the interpret function itself in the location-holding items.

The power of the generalized array lies in the simplicity of the descriptions of the data structures, independent of the specification of any algorithm to

interpret their structure.

- *Alternate Aggregrate Data-Structures.*

A number of different models of the generalized array have been proposed. Four typical ones will be discussed here, and one will be chosen.

[1] Many authors have suggested that the existing *APL* array can serve as a recursive data structure. Iverson[1], for example, has suggested the introduction of an enclose function, which would return a scalar encoding of its argument (something like the argument's Gödel number), and the inverse disclose function. Although these would allow the representation of all the above properties, the continual enclosing and disclosing is inconvenient for the study of the theory and so will be avoided. Nevertheless, such a scheme would be viable for implementation, particularly within the existing structure of *APL*.

[2] The recursive data structure of Gull and Jenkins[2] is one of the best known models of the generalized array. Unfortunately, their model does not directly support all of

[1] Iverson, K. E., personal communication, February 1980.

[2] Gull, W. E. and Jenkins, M. A., "Recursive Data Structures In *APL*," *Communications of the ACM 22*, 2(January 1979):79-96.

the principles of data. They define a generalized array in terms of its structural properties, instead of deriving it from the principles of data. Their system treats the empty array as a special case, thereby adding yet another consideration to the local complexity of their expressions. All of the items of their collections are restricted to be of the same type (their arrays are homogeneous), contrary to the principle of nesting.

[3] The recursive data structure of Ghandour and Mezei[1] is similar to More's arrays. Their system, however, is not axiomatic, does not present any extensions to More's work, and has not been as completely developed as More's has.

[4] The recursive data structure of More[2] supports the eight principles of data and the additional features discussed below. All references to generalized arrays will hereinafter refer to those of More unless otherwise qualified. The theory of generalized arrays presented here is based on version five of More's axiomatization, in which the errors in the previous four versions have been eliminated.

[1] Ghandour, Z. and Mezei, J., "Generalized Arrays, Operators and Functions," *IBM Journal of Research and Development 17*, 4(1973),355-352.

[2] See: More, T., "The Nested Rectangular Array As A Model Of Data", and the other works by More as cited in the bibliography.

Just as a set is not defined in axiomatic set theory, and a line is not defined in absolute geometry, a generalized array is not defined in More's axiomatic array theory. On the other hand, the axioms of the theory describe the effect of certain actions on certain objects. The interpretation of the objects as generalized arrays is an intuitive interpretation of the objects acted on by the axioms.

The axioms of More's theory describe the objects of a **standard**, **closed**, **one-sorted** theory. A standard theory is one in which all primitive operations are defined for all objects. A closed theory is one in which the application of any function to any object in the universe returns an object in the universe. A one-sorted theory is one in which there is only one domain of discourse.

More's notation is clear and concise and does not introduce a meta-notation for the discussion of the arrays and primitives (as Gull and Jenkins do). The notion of adjacency is maintained. The axioms apply without variation to all objects. There are no special cases such as a single kind of empty array (as there are in the systems of Gull and Jenkins and of Backus[1]

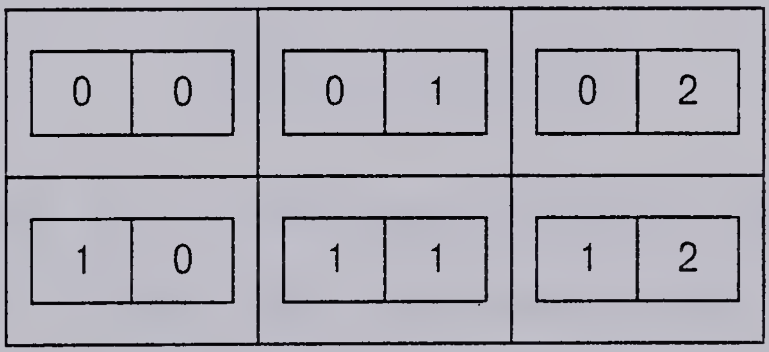
[1] Backus, J. W., "Can Programming Be Liberated...." Notice that the functions make a special case for the empty array and the undefined array.

), consequently, the exponential growth of special sub-cases in the proof of the meaning of an expression is avoided.

- *A Pictorial Representation of Generalized Arrays.*

For the purposes of the presentation of the theory of generalized arrays, in the rest of this chapter, the following pictorial representation of generalized arrays will be convenient. The representation is only able to represent arrays of up to two axes (although it can be extended) but that is all that is required for the purposes of this chapter.

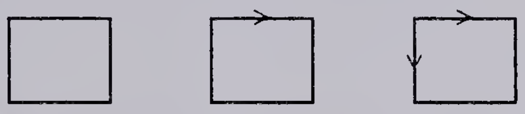
The structure of a generalized array will be indicated by a collection of boxes that contain the items held at the locations represented by the boxes. For example, the following diagram represents a particular two row by three column matrix of pairs.



0 0	0 1	0 2
1 0	1 1	1 2

[1] (cont'd)

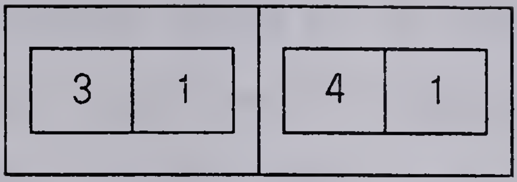
The number of axes in a diagram is usually apparent from the adjacency of the boxes in the diagram. In the previous example, there are clearly two axes, of length two and three. However, when an axis is of length one, a mechanism is needed to indicate the existence of the axis. In the following example, the array to the left has no axes, the array in the middle has one axis, and the array to the right has two axes.



The boxes that do not contain other boxes contain expressions. The value of the item in the box is the value of the expression. In the following vector, the first item is the numeric constant 24, the second item is the first character of the alphabet, and the third item is the value of the function *F* applied to the object *A*.

24	'*A*'	*F A*

The methods for the construction of these arrays will become apparent in the later sections of this thesis. Two special notations that will be particularly convenient are the scroll and the string. A **scroll** is a method for representing a suit of objects (which will be defined later) by juxtaposing the objects, seperated by blanks, within square brackets. The following two arrays are equivalent.



[[3 1][4 1]]

A **string** is a representation of a character constant as described in section 2.4. The scroll ['*A*' '*B*' '*C*'] is equivalent to the character string '*ABC*'. As will later be seen, since a scroll is a suit, the string '*A*' is equal to the mote which represents the first character of the alphabet.

3.2: An Axiomatic Theory of Generalized Arrays

The potential benefits of the axiomatization of a formal system are well known. More has axiomatized a **pure** theory of generalized arrays which parallels the axiomatization of the set theory of Zermelo, Skolem, and Frankel[1] Consequently, the mathematical derivations based on axiomatic set theory can be followed in the generalized-array theory in the parallel sense.

[1] See, for example, Suppes, P., *Axiomatic Set Theory*, (New York: Dover Publications Inc., 1972). The axiomatic equivalents are obtained from More, T., "Axioms and Theorems for a Theory of Arrays," *IBM Journal of Research and Development 17*, 2(March 1973):135-175 and More, T., "The Nested Rectangular Array as a Model of Data," p. 57-59.

The axiomatic theory, in addition to ordinary logic, requires the notions of the empty list of ordinal numbers, and the nine operations: equating, pairing, sublisting, reshaping, shaping, numerating, uniting, extracting, and replacement. These notions are related to the Zermelo-Skolem-Frankel axioms and the eight principles of data.

- *The Primitives of the Pure Theory.*

The nine primitive operations and their relations to the axioms of set theory and to the properties of data-structures are as defined as follows.

The primitive operation of **equality** corresponds to the axiom of extensionality and the principle of orientation. The $A=B$ equality of two arrays is true if and only if the arrays A and B hold the same items at the same locations and have the same orientation.

The primitive operation of **pairing** corresponds to the axiom of elementary classes and the principle of nesting. The **null** array θ is the empty list of the ordinal numbers. The null is primitive and corresponds to the empty set in set theory. The $A;B$ pairing of two arrays gives an array which holds A as its first item and B as its second item.

The primitive operation of **sublisting** corresponds to the axiom schema of separation and the principle of arrangement. The *A*\B sublisting of an array *B* by an array *A* is the one-axis array of the items of *B*, in main order, that correspond to the items of *A* which are true. The **main order** of the items of the array is obtained by increasing the last item in the vector of location positions most rapidly, the second last position only when the last position overflows the length of the last axis, and so on through the first position (with the initial location specified as the first location along all axes).

The primitive operation of **shaping** corresponds to the principle of valency, but does not correspond to any set theoretic axiom since set theory does not support the principle of valency. The ~*A* shape of an array *A* is a one-axis array holding the lengths of each axis of *A* according to the principles of valency and orientation. The shape of an array is said to **measure** the array.

The primitive operation of **reshaping** corresponds to the principle of repetition, but has no correspondence with a set theoretic axiom since set theory does not support the principle of repetition. The ~*A*ρ*B* reshaping of an array *B* to have the shape of *A* is an array with shape ~*A* whose items, in main order, are obtained, in main order, from *B*. If the number of items in *B* is less than the number of items in *A* then the items of *B* are repeated with the first item

following the last.

The primitive operation of **replacement** corresponds to the axiom of replacement. It applies to any collection. The replacement transform :Δ*A* of a monadic function Δ applied to an array *A* is the array obtained by replacing each item of *A* by the result of the application of Δ to the item. :Δ.*A*⍮*B* =. Δ*A*⍮Δ*B*.

The primitive operation **numerate** corresponds to the axiom of powers and the principle of smoothness. The ι~*A* numeration of an array ~*A* is an array of shape ~*A* in which each item is the **address** (or location specification) of itself, according to the principles of arrangement and orientation. By convention, the first item of a list will be at location zero. The address of an item therefore specifies the distance of the item from the origin of the axis, as opposed to the number of the item in the sequence along the axis.

The primitive operation **unite** corresponds to the axiom of unions and the principle of aggregration. The ∪*A* unite of an array *A* is an array of one axis which holds the items, in main order, of the items, in main order, of *A*.

The primitive operation **extract** corresponds to the axiom of choice and the property of well-ordering. The ⊃*A* extract of an array *A* is the first item of the main order of the items of *A*. *A* = ⊃. *A*⍮*B*.

- *Some Derivations from the Pure Theory.*

The **single** of a particular array is an array which holds as its single item the particular array. It is equal to the null reshaping of the particular array paired with itself:

$$\circ \equiv \theta\rho(\omega\overline{,}\omega).$$

The first item of the single of an array is the array itself: $\supset\circ A = A$.

The **ravel** of an array is a one-axis array or **list** of the elements of the array in main order. It is equal to the unite of the single of the array:

$$, \equiv \cup\circ\omega.$$

The **count** of a particular array is an array which contains as its single item the ordinal which corresponds to the number of items in the particular array. It is equal to the shape of the list of the array:

$$\# \equiv \sim,\omega.$$

Von Neumann's set-theoretic construction of the **ordinals** can be used to construct the ordinals in array theory as well:

$$\begin{aligned} 0 &\mathrel{=.} \#\theta \\ 1 &\mathrel{=.} \#\circ\theta \\ 2 &\mathrel{=.} \#(\theta\overline{,}\theta) \end{aligned}$$

The value **false** can be represented by the ordinal zero and the value **true** can be represented by the other ordinals.

Arithmetic and logical operations can be defined in the same manner as they are in set theory or in *APL*, for example:

$$+ \equiv \#\cup.\alpha;\omega$$

$$\neg \equiv 0{=}\omega$$

$$\vee \equiv \alpha{+}\omega.$$

In the pure theory there are no possible errors, all functions are defined for all objects.

- *The Composition Transform.*

The juxtaposition of two or more functions implies the **composition** of those functions according to the scope rules of Chapter Two. For example, the list function is defined to be the composition of the unite and single functions applied to a right argument: $,\equiv\cup\circ\omega$. The introduction of explicit compose and apply names is not required here, since their invocation is implied by the juxtaposition of functions and objects.

The function $\bar{\alpha}\bar{\omega}$ is the composition transform of the two functions $\bar{\alpha}$ and $\bar{\omega}$. The ɪ'$\bar{\alpha}\bar{\omega}\omega$' meaning of the function which is the composition of (the right-monadic functions) $\bar{\alpha}$ and $\bar{\omega}$ as applied to (the right argument) ω is: the object which is

the result of the application of $\bar{\alpha}$ to the object which is the result of the application of $\bar{\omega}$ to the object ω. In the list example, unite is applied to the application of shape to ω.

The composition operation can be defined in terms of application. Using $\circ$ to denote composition and : to denote application (for the remainder of this sentence only): $\circ \equiv \bar{\alpha}:(\bar{\omega}:\omega)$. Application is obtainable from the pure theory following the set theoretic definition of relative product.[1]

In his development of the pure array theory, More has generated a composition table for pairs of primitive and certain derived functions.[2] The table is a particularly useful collection of identities, in which theorems like $\upsilon:\upsilon\circ\circ\omega=,\omega$ can be found. Although the table will not be reproduced here, its study is particularly interesting to a student of the theory. Many of the paradoxes in the first four versions of More's theory were discovered with the aid of the table.

[1] See Suppes, "Axiomatic Set Theory," p. 63. The relative product of two relations is defined therein as $A/B=\{\langle x,y\rangle: (\exists z)(xAz \;\&\; zBy)\}$.

[2] More, T., "On the Composition of Array-Theoretic Operations," *Technical Report G320-2113*, (Cambridge: IBM Scientific Center, May 1976)

3.3: The Applied Theory

- *Individuals Are Included as Motes.*

Although the pure theory is complete, it is difficult to use because of its lack of elemental data items. In order to obtain the **applied** theory, Quine's[1] representation of an **individual** is added to the pure theory. An individual is represented by a **mote**, a zero axis array which holds itself as its single item[2] Consequently, A is a mote if and only if $A{=}\circ A$. Furthermore, A is a mote if and only if $A{=}\supset A$:

$$\begin{array}{lll} \circ A = A & \leftrightarrow \ \supset\circ A = \supset A & \qquad\qquad A = B \rightarrow \Delta A = \Delta B \\ & \leftrightarrow \ A = \supset A & \qquad\qquad \supset\circ A = A \end{array}$$

The value of a numeric formal symbol is the mote which represents the number represented by the formal symbol. The value of a character string is a suit (to be defined later) in which each item is the mote whose value is the character represented in the corresponding location in the string. The ⍰ is the **cipher**, a mote which is neither a number nor a character. The cipher is usually interpreted as meaning "undefined." The ○ is the mote which represents logical

[1] Quine, W. V., "Unification of Universes in Set Theory," *Journal of Symbolic Logic 21*, (1956):267-279.

[2] More, T., "Notes on the Development of a Theory of Arrays," *Technical Report 320-3016*, (Philadelphia: IBM Scientific Center, May 1978), p. 26-33.

falsity, or $\theta=\theta=\theta$, and $\perp$ is the mote which represents logical **truth**, or $\theta=\theta$. Additional types of motes can be added to the theory as necessary (for example, complex or rational values).

The treatment of individuals as motes has several advantages. Since motes are not distinct from arrays, the theory remains one-sorted. There is no need for a primitive mote-predicate like ATOM in LISP, since the moteness of an array A can be tested with the expression $A=\circ A$. The intuitive relation $\supset\supset\ldots\supset 1=1$ is preserved when 1 is a mote.

Although the notion of self-membership appears difficult to comprehend in terms of physical objects (a box, for example, cannot contain itself), it is the single property that, in all cases, identifies classes containing only a single member (individuals) from all other classes. The class of diamonds is not an individual because it contains both the class of the Hope diamond and the class of the Pink Panther diamond. But the class of the Pink Panther diamond is an individual because the only class it contains is the class of the Pink Panther diamond, that is: itself. Similarly, a mote is a generalized array containing only itself.

The motes of the applied theory represent the numbers and characters that form the data we wish to manipulate.

- *Extending the Pure Theory.*

The applied theory requires the standard notions of logic, the universe of motes as described above, and the primitive functions described below. The primitive functions of the applied theory are those of the pure theory plust those functions which transform motes into other motes, such as the arithmetic operations.

A number of useful derived functions will also be presented herein. The interpretation of the function and its useful properties will be discussed in each case. Additional terminology will be introduced where necessary.

- *Array Structure and Pervasive Functions.*

Two arrays have the same **structure** when they are of the same shape and all of their items are of the same structure. Clearly, motes are of the same structure. A function ∆ is said to be **pervasive** when :∆*A*=∆*A*. Such a function returns an array of the same structure as its argument array in which each mote is replaced by the result of the application of ∆ to the mote. For example, if ∆ is pervasive then ∆.1⍮2⍮.3⍮4 is equal to ∆(1⍮2)⍮∆(3⍮4) which is equal to (∆1⍮∆2)⍮(∆3⍮∆4). If pervasive operations are to work for numbers, then each number must be treated as an individual rather than as a collection of digits. Consequently, numbers are motes themselves.

On the other hand, the treatment of a string of characters as a mote means that there is no way to take apart the string for operations on the individual characters, other than to introduce primitive character separation functions like en-mote and de-mote. Instead, the applied theory treats a string of characters as a list of motes.

- *Equality, Shape, Reshape, and Vacate.*

The *A*=*B* **equality** of two arrays *A* and *B* is logically true if and only if *A* and *B* hold the equal items at the same locations and have the same orientation.

The ~*A* **shape** of and array *A* is a **suit** holding the lengths of the axes of *A*. A suit is an **apovalent** array: it has the minimum number of axes necessary to hold its items. The shape of a no-axis array is the empty list of ordinals, the shape of a one-axis array is the mote holding the ordinal that measures the length of the axis, and the shape of a two or more axis array is a one-axis array of the ordinal lengths of the axes. A suit of ordinals is a **primary** array.

$$\begin{array}{|c|c|}\hline 2 & 2 \\ \hline\end{array} = \sim \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & 0 \\ \hline\end{array} \quad \text{and} \quad \begin{array}{|c|}\hline 3 \\ \hline\end{array} = \sim \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline\end{array}$$

The ~*A*ρ*B* **reshape** of an array *B* to have the shape ~*A* is an array of the same shape as *A* in which the items in main order are selected from the items (in main order) of *B*. Consequently: ~(~*A*ρ*B*)=~*A*.

$$\begin{array}{|c|c|c|}\hline 0&1&2\\\hline 3&0&1\\\hline 2&3&0\\\hline\end{array} = \begin{array}{|c|c|}\hline 3&3\\\hline\end{array} \;\rho\; \begin{array}{|c|c|c|c|}\hline 0&1&2&3\\\hline\end{array}$$

The \A **vacate** of an array *A* is obtained by selecting no items from *A*.

$$\backslash \equiv 0\rho\omega.$$

The vacate of an array is an array of one axis holding no items. Consequently, 0=~\A. Any array holding no items is said to be **empty**.

The θ empty array obtained by vacating an ordinal is the **null**.

$$\theta \equiv \backslash 0.$$

The null reshaping of an array is an array of shape null. Since the shape of an array is a suit of the lengths of the axes, the null reshaping of an array *A* is an array of no axes, the items of which are selected from *A* in main order. But the array θρ*A* of no axes can hold at most one item, the first item in the main order of *A*. The address of the item is a suit of the locations of the item with respect to each axis, but since there are no axes the address is the

null, the empty list of ordinals specifying the location on each axis.

A mote can be metaphorically thought of as an item surrounded by an infinite number of infinitely thin skins. Any number of these skins can be added or removed, and there will still be an item surrounded by an infinite number of infinitely thin skins. The onion has no axes, but can still hold a single item at its center. However, as soon as two or more of these onions are collected together, at least one axis of orientation is required to distinguish their locations.

- *Pair, Single, and Sublist.*

The $A;B$ **pair** of the array A with the array B is an array which holds as its first item the array A and which holds as its second item the array B. Consequently, the array $A;B$ is a one-axis array of two items: $2=\sim(A;B)$. Any one-axis array with two items is a pair.

The $\circ A$ **single** of an array A is defined to be the no-axis array which holds A as its single item.

$$\circ \equiv \theta\rho(\omega;\omega) \quad \text{and} \quad \theta=\sim\circ A.$$

Any no-axis array is a single. Any one-item array is **singular**.

$$\circ . A \bar{;} B = \boxed{\begin{array}{|c|c|}\hline A & B \\ \hline \end{array}}$$

It is very important to remember that for any mote *M*, *M*=⊃*M* and *M*=∘*M*. This applies for all the numbers and characters, since they are motes.

$$0 \text{ equals } \boxed{0} \text{ equals } \boxed{\boxed{0}}$$

The *A*\B **sublist** of an array *B* by an array *A* is a one-axis array of the items of *B* (in main order) which correspond to logically true items in the main order of ~*B*ρ*A*. The count of the sublist is equal to the number of items in ~*B*ρ*A* which are true. Vacate is the same as the everywhere-false sublisting: \A=○\A.

$$\begin{array}{|c|c|}\hline 3.4 & 5.6 \\ \hline \end{array} = [\circ\ \perp\ \perp\ \circ]\ \backslash\ \begin{array}{|c|c|c|c|}\hline 1.2 & 3.4 & 5.6 & 7.8 \\ \hline \end{array}$$

- *Unite, Union, and Suit.*

The ∪*A* **unite** of an array *A* is a one-axis array which holds (in main order) the items of the items of *A*.

$$\cup(\circ A \bar{;} \circ B) =.\ A \bar{;} B.$$

[0 0 0 1 0 2 1 0 1 1 1 2] = ∪

[0 0]	[0 1]	[0 2]
[1 0]	[1 1]	[1 2]

The *A*∪*B* **union** of two arrays *A* and *B* is a one-axis array of the items of *A* in main order followed by the items of *B* in main order:

∪ ≡ ∪.α;̄ω and ∘*A*∪∘*B* =. *A*;̄*B*.

The ;̄*A* **suit** of an array *A* is an array of the least possible number of axes holding the items of *A* in main order. *A* is a suit if and only if ;̄*A*=*A*.

- *Extract, Tail, Last, and List.*

The ⊃*A* **extract** of an array *A* is the single which holds the first item in the main order of *A*. The extract of the single of an array is the array itself.

⊃∘*A*=*A* and ⊃(*A*;̄*B*)=*A*.

The ł*A* **tail** of an array *A* is the one-axis array which holds all the items of *A*, in main order, except the first.

ł≡ ○∪(~ωρ⊥)\ω.

ł∘*A*=\∘*A* and (ł.*A*;̄*B*) =. 1ρ∘*B*.

The ⊂*A* **last** of an array *A* is the last of the items of *A* in main order:

⊂ ≡ ⊃.ł(~ωρ○∪⊥)\ω

⊂*A*=*A* ⊂∘*A*=*A* (⊂.*A*;̄*B*)=*B*

The ,A **list** of an array A is a one-axis array which holds the items of A in main order. Therefore, list can be defined as the union of the single which contains the array or as the everywhere-true sublist:

, ≡ ∪∘ω and , ≡ ⊥\ω

Any one-axis array is a list. A function Δ is **idempotent** if and only if ΔΔA=ΔA. Listing is idempotent: ,,A=,A. A **solitary** is a list which holds exactly one item. A is a list when ,A=A. A is a solitary when ~A=1.

- *Count, Valency, and Link.*

The #A **count** of an array A is equal to the number of items in A. The number of items in an array is equal to the number of items in the list of the array. The number of items in a list is equal to the shape of the list. Therefore, count can be defined to be the shape of the list of A:

≡ ~,A.

0=#θ. 1=#∘A. 2=#A⍮B. #,A=~,A.

The **valency** (number of axes) of an array is equal to the count of the shape of the array.

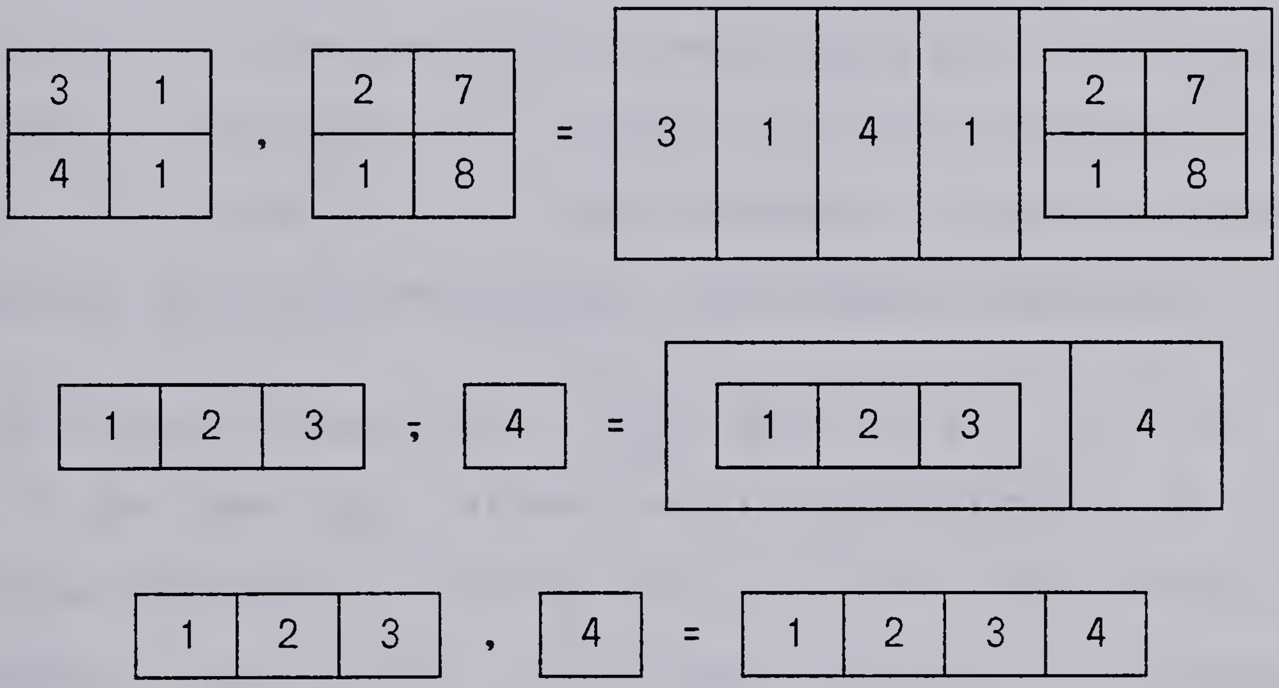
0=#~∘A. 1=#~.A⍮B. 1=#~\A.

These definitions of count and valency provoked the earlier decision to make the result of shape a suit instead of a list. Since the count and valency of an array are

ordinal numbers, they should be motes. If shape yielded a list, then count and valency would be solitaires instead of motes. Instead, since shape yields a suit, count and valency yield singles containing motes, which are by definition the motes themselves.

The *A*,*B* **link** of an array *A* with an array *B* is a list of the items of *A* in main order followed by *B* itself:

$$, \equiv \alpha\cup\circ\omega$$



The following arrays are equal:

$$A\bar{;}B \quad \circ A\cup\circ B \quad \circ A,B \quad ,\circ A,B \quad \theta,A,B.$$

- *Type and Prototype.*

Since this is a standard theory, the object ⊃\A must be an array. But what is the first item of the empty array \A? Suppose that *I* is a list of integers. If *I* has more than one item, then ⊃∤*I* will be an integer. In the interests of

generality, even if *I* holds only one integer, ∍⍀*I* should be an integer. However, if *I* holds only a single integer, then ⍀*I*=\I, so that ∍\I should be an integer. Moreover, if *B* is an array obtained by vacating *A*, then the extract of *B* should be typical of the items of *A*.

The object ∍\A has received considerable attention in version five of More's theory[1] In order to solve the problem, the τ*A* **type** of an array *A* has been formalized. Intuitively, the type of an array is an array of the same structure as the array, in which each mote is replaced by its type. The type of an integer is the archetype 0, the type of a truth value is the archetype ○, and the type of a character is the archetype 'b̸' (the space character).

An array is **monotypic** if and only if all of its items are of the same type, otherwise it is **polytypic**. An array *A* **typifies** and array *B* if and only if *A*=τ*B*. The arrays *A* and *B* **depict** each other if and only if τ*A*=τ*B*. An array *A* is **typical** if and only if *A*=τ*A*. If an array is monotypic then the type of the first item of the array typifies all the other items of the array. No item of a polytypic array typifies all the other items in the array. The τ∍*A* type of the first item of an array is the **prototype** of the array.

[1] More, "The Nested Rectangular Array as a Model of Data," p. 63-65.

The type primitive must satisfy four properties. First, the type operation must be pervasive, for types are only defined for motes. Second, the type of a typical mote must equal the typical mote, so the type primitive must be idempotent.

Third, the type of an array A is an array of the same structure as the array A holding the type of each of the items of the array. But since the empty array holds no items, an array of the same structure holding the types of the items of the empty array must also hold no items. The type of an empty array $\backslash A$ must be an empty array of the same structure as $\backslash A$, so that for an empty array $\backslash A$, $\tau\backslash A=\backslash A$, and $\backslash A$ is vacuously typical. But if an array is typical then obviously each item of the array must be typical, so that the first item of an empty array must be typical.

Fourth, by previous argument, it is desireable to have the first item of an empty array typify the items that the array held before being emptied. Since the prototype of a monotypic array typifies the entire array, and since an empty array is vacuously monotypic, the prototype of an empty array $\backslash A$ should equal the type of the first item of the array A.

In summary, the type primitive should satisfy the following four properties:

$\tau A = :\tau A$ pervasive conjecture

$\tau A = \tau\tau A$ idempotent conjecture

$\supset\backslash A = \tau\supset\backslash A$ typical conjecture

$\tau\supset\backslash A = \tau\supset A$ prototype conjecture

The typical-conjecture states that the extract of an empty array is equal to the prototype of the empty array. Therefore, even though an empty array does not actually hold any items, it must retain its prototype.

From the typical-conjecture, it folllows that

$$\supset\backslash\circ A = \tau\supset\backslash\circ A$$

and by the propotype-conjecture

$$\tau\supset\backslash\circ A = \tau\supset\circ A$$

so that

$$\supset\backslash\circ A = \tau A.$$

In view of these results, the definition

$$\tau \equiv :(\supset\backslash\circ)\omega$$

and the **prototype axiom**

$$\supset\backslash A = \tau\supset A$$

are introduced. The definition states that the type of an array is equal to the replacement transform of the extract of the vacate of the single the array. The axiom states that the extract of the vacate of an array is equal to the type of the first item the array. This is clearly a recursive definition of the type function.

For any mote *M*, *M*=∘*M* and *M*=⊃*M*. For any function ∆, :∆∘*A*=∘∆*A*. So, for *M* a mote, ⊤*M* is equal to :⊤*M* because ⊤ is pervasive, which is equal to :⊤∘*M* because *M* is mote, which is equal to ∘⊤*M*. Since ⊤*M*=∘⊤*M* it follows that when *M* is a mote, ⊤*M* is a mote.

The pervasion of the type function **converges** at motes. The example in Figure 3.3-1 illustrates the process.

```
⊤.(1⍮2)⍮(3⍮4) ←→ :(⊃\∘). (1⍮2)⍮(3⍮4)              ⊤≡:(⊃\∘)
              ←→ ⊃\∘(1⍮2) ⍮ ⊃\∘(3⍮4)              def'n of :
              ←→ ⊤⊃∘(1⍮2) ⍮ ⊤⊃∘(3⍮4)                ⊃\A=⊤⊃A
              ←→ ⊤(1⍮2) ⍮ ⊤(3⍮4)                       ⊃∘A=A
              ←→ :(⊃\∘)(1⍮2) ⍮ :(⊃\∘)(3⍮4)         ⊤≡:(⊃\∘)
              ←→ ⊃\∘1⍮⊃\∘2 ⍮. ⊃\∘3⍮⊃\∘4           def'n of :
              ←→ ⊃\1⍮⊃\2 ⍮. ⊃\3⍮⊃\4                     Motes
              ←→ ⊤⊃1⍮⊤⊃2 ⍮. ⊤⊃3⍮⊤⊃4                 ⊃\A=⊤⊃A
              ←→ ⊤1⍮⊤2 ⍮. ⊤3⍮⊤4                         Motes
              ←→ (0⍮0) ⍮ (0⍮0)                      Mote type
```

Figure 3.3-1: Illustration of Type

- *Augmented Values, Empty Lists, and Void.*

The definition of a truth value is extended for convenience, to **augmented** truth values, as follows. Any typical mote is equivalent to the ○ truth value false. Any other mote is equivalent to the ⊥ truth value true. All of the functions which expect a truth value for an argument are extended to accept augumented truth values. In particular,

the integers zero and one now correspond to the truth values false and true respectively.

Analogous to the augmented truth values are augmented numeric values. The ○ truth value is equivalent to 0 as an augmented numeric value and the ⊥ truth value is equivalent to 1 as an augmented numeric value.

As previously defined, the null is the empty list of the ordinals: θ≡\0. The ⌽ **nil** is the empty list of characters: ⌽≡\'⍉' which is equal to ''. The ẋ **nix** is the empty list of ciphers: ẋ≡\⍰.

In order to obtain an empty list with prototype ⊤*A* instead of ⊤⊃*A* it is necessary to empty the single containing *A*. The \∘*A* **void** of an array *A* is the empty list with prototype ⊤*A*.

- *The Replacement, Positional, and Each Transforms.*

The **replacement** transform of the applied theory can now be defined:

: ≡ ~ωρ. ∘ω̄⊃ω ∪ :ω̄∤ω

This recursive definition reads as: the result of the application of :Δ to *A* is obtained by reshaping to shape ~*A* the union of ∘Δ⊃*A* with the application of :Δ to ∤*A*.

In order to terminate this recursive definition, the effect of the replacement transform on an empty list must be determined.

```
:Δ\A ←→ ~\Aρ. ∘Δ⊃\A ∪ :Δǂ\A            def'n of :
     ←→ 0ρ. ∘Δ⊃\A ∪ :Δǂ\A                    0=~\A
     ←→ \∘Δ⊃\A                         (0ρ.∘A∪B)=\∘A
     ←→ \∘ΔT⊃A                                ⊃\A=T⊃A

T⊃:Δ\A ←→ T⊃\∘ΔT⊃A                      above result
       ←→ ⊃\\∘ΔT⊃A                            T⊃A=⊃\A
       ←→ ⊃\∘ΔT⊃A                              \\A=\A
       ←→ T⊃∘ΔT⊃A                             T⊃A=⊃\A
       ←→ TΔT⊃A                                 ⊃∘A=A
```

Consequently, the application of :Δ to an empty list \A is an empty list, the prototype of which is the type of the application of Δ to the prototype of *A*. :Δ\A is an empty array with prototype TΔT⊃A.

Notice that the above definition of the replacement transform does not use any control structure such as the conditional if-then-else structure. It is a simple recursive definition which embodies its own termination criteria, thereby preserving the intent of section 2.1. The recursion terminates at empty lists because the result of the application of :Δ to an empty list is an empty list, and because ∘A∪\B=,A according to the definition of union.

Corresponding to the monadic replacement-transform of a monadic function is the dyadic positional-transform of a dyadic function. If *A* and *B* are the same shape, then the A:ΔB positional transform of Δ is defined to be

: ≡ ~αρ. ∘(⊃α ω̄ ⊃ω) ∪ (ǂα :ω̄ ǂω).

For empty arrays \A and \B the shapes are 0=~\A and 0=~\B and so, following the proof of reduction on an empty array:

\A:Δ\B = \(⊤⊃A Δ ⊤⊃B)

Consequently, the positional-Δ transform on two empty arrays A and B is an empty array for which the prototype is the prototype of the result of Δ on the prototypes of A and B. For non-empty arguments the recursion continues until ⍭α and ⍭ω are empty and the resulting empty list disappears in the union operation, just as it did for the replacement transform.

The above definition of the positional-transform actually applies to all argument arrays independent of their shape. The result of A:ΔB will always be of shape ~A in which the items of A (in main order) are paired with the items of B (in main order). Consequently, the result of ∘A:ΔB will be ∘.AΔ⊃B but the result of A:Δ∘B will be an array of shape ~A in which the first item is ⊃AΔB and the J′th item of A is replaced by JΔ\B.

In view of the above positional-transform result, there are dyadic transforms for combining a left argument with each item of a right argument and for combining a right argument with each item of a left argument. The A⊂̈ΔB **each-right** transform of Δ on A and B is an array of shape ~B in which each item I of B is replaced by AΔI:

$$\ddot{\subset}\ \equiv\ (\sim\omega\rho\circ\alpha)\ :\bar{\omega}\ \omega.$$

$$\overset{\rightarrow}{\boxed{2}}\ \ddot{\subset}\Delta\ \boxed{6 \,\big|\, 9}\ =\ \boxed{\overset{\rightarrow}{\boxed{2}}\ \Delta\ \boxed{6}\ \Big|\ \overset{\rightarrow}{\boxed{2}}\ \Delta\ \boxed{9}}$$

Similarly, the $A\ddot{\supset}\Delta B$ **each-left** transform of Δ on A and B is an array of shape $\sim A$ in which each item I of A is replaced by $I\Delta B$:

$$\ddot{\supset}\ \equiv\ \alpha\ :\bar{\omega}\ (\sim\alpha\rho\circ\omega).$$

Consider the following special cases for the each-right and each-left transforms (as illustrated with the each-right transform). If the right argument is empty then:

$$\begin{aligned} A\ddot{\subset}\Delta\backslash B &\leftrightarrow (0\rho\circ A):\Delta\backslash B \\ &\leftrightarrow \backslash(\top\supset\circ A\ \ \Delta\ \ \top\supset B) \\ &\leftrightarrow \backslash(\top A\ \ \Delta\ \ \top\supset B) \end{aligned}$$

Consequently, the each-right transform of Δ with the left argument A and the empty right argument $\backslash B$ is an empty list with prototype equal to the prototype of $\top A\ \Delta\ \top\supset B$.

On the other hand, if the left argument is empty then the result is just an array of shape $\sim B$ in which each item I of B is replaced by $\backslash A\Delta I$.

- *The Reduction Transform.*

Suppose that L is the list $A\bar{,}B,C,D$ containing the four items A, B, C, and D. Intuitively, the $\overset{\wedge}{,}\Delta$ **reduction** transform of the dyadic function Δ should have the following

effect:

♤ΔL =. AΔBΔCΔD.

The obvious recursive definition:

♤ ≡ ⊃ω ω̄ ♤ω̄ ⍭ω

along with the **reduction axiom**

♤Δ(A⍪B) =. AΔB

is sufficient except for the following implications:

♤Δ∘A =. A Δ ⊤A Δ ⊤A Δ ...

♤Δ\A =. ⊤⊃A Δ ⊤⊃A Δ ⊤⊃A Δ ... .

Intuitively, if + is the dyadic addition function then

♤+3 =. 3 + 0 + 0 + ...

which is equal to 3. But for the union function

♤∪3 =. 3 ∪ 0 ∪ 0 ∪ ...

which is equal to three union the infinite list of zeroes.

In *APL*, the problem is solved by having f/1ρ*G* equal 1ρ*G* itself, and by having f/0ρ*G* equal *z*, the right-zero of f (*x*f*z* = *x*). This is less than optimal, since ∧/55 is equal to 55, and the right zero for many defined functions is unknown. However, it does support the intuitively satisfying relationship

♤Δ(A⍪B) =. ♤Δ∘A Δ ♤Δ∘B

Alternately, the value of ♤Δ∘A and ♤Δ\A can both be taken to be ⍰, or undefined.

The non-termination of the recursive definition of a function, for a particular value, does not necessarily imply that the function is undefined at that value. Some other defintion of the function might terminate for that value. The previous definition of reduction does not terminate for singles and empty arrays, but some other definition of reduction might. At this point however, there appears to be no way to reconcile the underlying question: "what is the effect of a dyadic function applied to a single argument?", without being able to find the right-zero of a function. Perhaps a future result will improve this situation.

- *Numerate, Enumerate, and Addresses.*

The ⍳~*A* **numerate** of an array ~*A* is an array of shape ~*A* in which each item is the address of itself. If *I* is an item in ⍳~*A* then *I* will be at location *I* in ⍳~*A*. Alternately, ⍳~*A* is known as the **enumerate** of an array *A*. Numerate is only defined for arguments which are valid shapes. If there is no array *A* such that *B*=~*A* then ⍳*B*=⍰.

The first item of ⍳~*A* is the address of the first item of *A*, which is #~*A*ρ0. But that is exactly equal to a suit with a zero for each item in ~*A*, so ⊃⍳~*A*=⊤~*A*. The prototype, or type of the first item, of ⍳~*A* is the type of an address in *A* and so must depict an address in *A*. But ⊃⍳~*A*=⊤~*A* so ⊤⊃⍳~*A*=⊤⊤~*A* which is ⊤~*A* because ⊤ is idempotent. Consequently, ⊤~*A* is both the extract and the prototype of

ι~*A*.

If 2⍪3 = ~*A* then *A* is an array of two rows and three columns. The enumerate of *A* is equal to the numerate of the shape of *A* which is equal to ι.2⍪3. The result must be a two by three table, each item of which is its own address. Consequently:

$$\iota\ \begin{array}{|c|c|}\hline 2 & 3 \\ \hline \end{array} \ = \ \begin{array}{|c|c|c|}\hline \begin{array}{|c|c|}\hline 0 & 0 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 0 & 1 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 0 & 2 \\ \hline \end{array} \\ \hline \begin{array}{|c|c|}\hline 1 & 0 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 1 & 1 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array} \\ \hline \end{array}$$

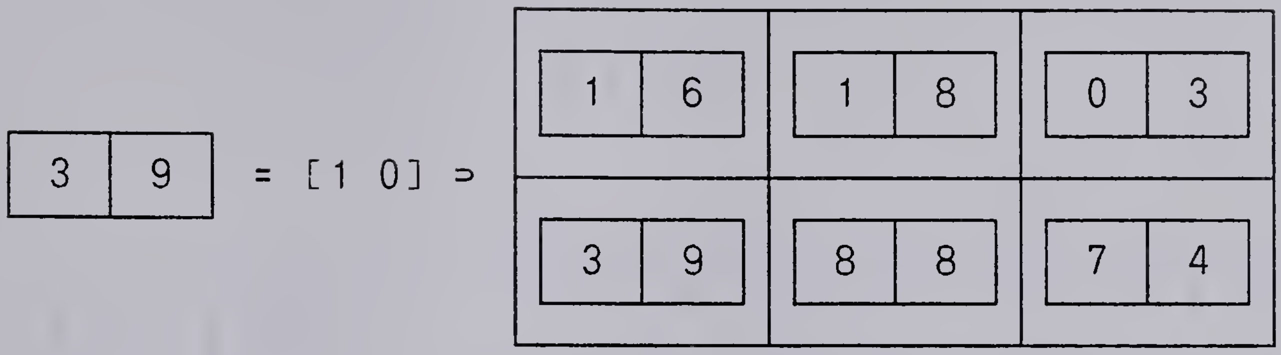
The shape of ι*A* is equal to *A*. ι1 is an array of shape 1 holding 0 at location 0: ι1 = ,0. ι0 is an array of shape 0 which must therefore hold no items. But it must also hold the addresses of its items, therefore ι0 = θ. ιθ is an array of shape θ which means that ιθ is a single. The single item of ιθ must be the address of itself. Consequently, ιθ = ∘θ.

- *Pick, Box, and Projection.*

The *A*⊃*B* pick of an array *B* by an array *A* is the item of *B* which occurs at address *A*.

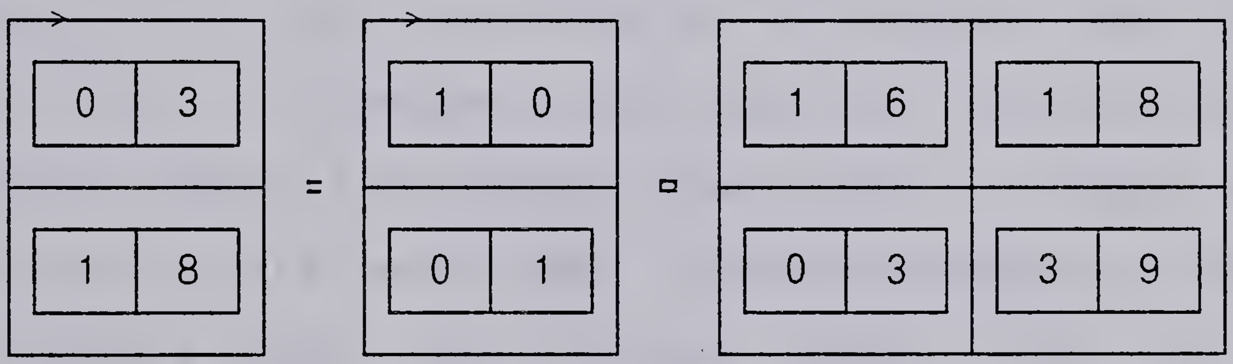
$$\supset \ \equiv \ \supset(\iota\sim\omega\ddot{\supset}=\alpha\backslash\omega)$$

When *A* is not an address of *B* the left argument of the sub-list function will be all ○ and so *A*⊃*B* will equal ⊃\B, the prototype of *B*.



The *A*▭*B* **box** of an array *B* by an array *A* is an array of shape ~*A* in which each item is the item of *B* picked by the corresponding item in *A*:

▭ ≡ α⊃̈⊃ω



The items of *A* in *A*▭*B* are addresses for *B*. If an item of *A* is not an address of *B*, then the corresponding item of *A*▭*B* is, by the definition of pick, equal to ⊃*B*, the prototype of *B*. If *A* is a mote then *A*▭*B* = ∘(*A*⊃*B*).

```
A  =  ∘A
A  ▭  B    ←→  A ⊃̈⊃ B                               def'n of ▭
           ←→  ~A  ρ.  ∘(⊃A  ⊃  B) ...              def'n of ⊃̈
           ←→  θρ  ∘(⊃A  ⊃  B)                          θ = ~∘A
           ←→  θρ  ∘(A⊃B)                               ⊃∘A = A
           ←→  ∘(A⊃B)                                 θρ∘X = X
```

The *A*±*B* **projection** of an array *A* on an array *B* is an array of shape ~*B* in which each item *I* of *B* is replaced by the item at address *A* of *I*. *A* is an address for the *items* of

B.

$$\pm \equiv \alpha\ddot{\subset}\supset\omega.$$

$$\begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline \end{array} = 1 \pm \begin{array}{|c|c|c|}\hline \begin{array}{|c|c|}\hline 0 & 0 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 0 & 1 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 0 & 2 \\ \hline \end{array} \\ \hline \begin{array}{|c|c|}\hline 1 & 0 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 1 & 1 \\ \hline \end{array} & \begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array} \\ \hline \end{array}$$

The above definitions of pick, box, and project illustrate a very powerful technique. The pick function is defined only for the extraction of a single item from a single array. Extension by the each-left transform defines a function which selects many items from a single array. Extension by the each-right transform defines a function which picks a single item from many items. Each of these box and project functions can further be extended with the each-transforms to generate more complicated operations. This theme will reappear in the definition of many of the more complex functions of array theory.

- *Locate, Seek, and Find.*

The $A\circ B$ locate of *A* in an array *B* is a list of the addresses of all the occurrences of *A* in *B*:

$$\circ \equiv \alpha\ddot{\subset}=\omega \setminus \iota\sim\omega$$

If *A* does not occur in *B*, which is the case when *B* is empty, then $A\circ B=\setminus\circ\sim B$.

$$
\begin{array}{lll}
A\circ\backslash B & \leftrightarrow\ A\ \ddot{\subset}{=}\backslash B\ \ \backslash\iota\sim B & \text{def'n of } \circ \\
 & \leftrightarrow\ \backslash(\tau A\ =\ \tau\supset B)\ \backslash\ \ \iota\sim B & A\ddot{\subset}\Delta\backslash B=\backslash(\tau A\Delta\tau\supset B) \\
 & \leftrightarrow\ \backslash\iota\sim B & \text{def'n of } \backslash \\
 & \leftrightarrow\ \backslash\circ\supset\iota\sim B & \text{def'n of } \backslash \\
 & \leftrightarrow\ \backslash\circ\sim B & \tau\supset X=\tau\supset Y\ \rightarrow\ \backslash X=\backslash Y
\end{array}
$$

$A\circ\backslash B$ is empty and has as prototype the type of the shape of B. $A\circ\backslash B$ is the empty list of addresses for B.

The $A\exists B$ **seek** for A in B is a primary array containing the address of the first occurrence of A in B. If A does not occur in B then $A\exists B$ is equal to $\sim B$. Seek is defined to be

$$\exists\ \equiv\ \supset(\alpha\circ\omega,\ \sim\omega)$$

Notice that $A\neq B\ \rightarrow\ A\exists\circ B=\theta$.

The $A\iota B$ **find** of an array A in an array B is an array of shape A in which each item in A is replaced by the address of its first occurrence in B. If a particular item of A does not occur in B then the item is replaced by $\sim B$. The definition of find is

$$\iota\ \equiv\ \alpha\ddot{\supset}\exists\omega.$$

$$
\begin{array}{|c|c|}\hline [0\ \ 1] & [2\ \ 2] \\ \hline\end{array}\ =\ [\text{'}T\text{'}\ \ \text{'}O\text{'}]\ \iota\ \begin{array}{|c|c|}\hline \text{'}AR\text{'} & \text{'}T\text{'} \\ \hline \text{'}FU\text{'} & \text{'}L\text{'} \\ \hline\end{array}
$$

Again, we see the power of the each transforms in the definition of the find function in terms of the seek function, and in the definition of the seek function in terms of the locate function. This same mechanism appears in the following definition of deletion in terms of

membership, and of membership in terms of occurrence.

- *Occurrence, Membership, Deletion, and Seperation.*

The $A\bar{\epsilon}B$ **occurrence** of an array A in an array B is $\perp$ if and only if there is an item in B which is equal to A. An array A occurs in an array B only if the list of addresses of A in B is not empty, consequently:

$$\bar{\epsilon} \equiv \circ =.\ 0 =.\ \#\alpha\circ\omega.$$

The $A\epsilon B$ **membership** of the items of the array A in the array B is an array of shape A in which each item of A is replaced by the true or false value of the predicate "the item occurs in B." Consequently,

$$\epsilon \equiv \alpha\ddot{\supset}\bar{\epsilon}\omega$$

The $A\ast\!\!\!\!|\,B$ **deletion** of the items of an array B by the items of an array A is the list of all the items of B that are not picked by the addresses in A.

$$\ast\!\!\!\!| \equiv \circ\ddot{\subset}=(\iota\sim\omega\ddot{\supset}\epsilon\alpha)\backslash\omega.$$

Consequently, $A\ast\!\!\!\!|\,B$ is the sublist of all the items of B that remain after each item of B selected by an address in A is deleted.

The $\underline{\times}\Delta A$ **separation** of an array A by a monadic predicate Δ is the sublist of all the items of A that correspond to true objects in $:\Delta A$.

$$\underline{\times} \equiv :\bar{\omega}\omega\backslash\omega$$

- *The Equal Predicate and Uniformity.*

The =*A* **equal** predicate is true if and only if all of the items of *A* are equal to each other, which is the case if and only if all of the items are equal to the first item.

```
= ≡ ○ =. 0 ϵ̄. ⊃ω ⊂̈= ω
```

An array is **unishape** if and only if each item in *A* is of the same shape as every other item in *A*. Therefore, an array *A* is unishape if and only if =:~*A*. The ~⊃*A* **protoshape** of an array *A* is the shape of the first item of *A*. An array *A* is **univalent** if and only if each item of *A* has the same valence as every other item in *A*. Therefore, an array *A* is univalent if and only if =:(#~)*A*. An array is **unitype** if and only if the type of each item of *A* is equal to the type of every other item in *A*. Therefore, and array *A* is unitype if and only if =:τ*A*

- *Trim, Form, and Pack.*

The τ*A* **trim** of an array *A* is obtained from *A* as follows. First, *A* is made univalent by appending **singular** axes (axes of length one) to the shape of *A*. Second, the corresponding non-singular axes of the items of the result of the first step are made to be the same length by truncating all corresponding non-singular axes to the length of the shortest corresponding non-singular axis. Third, each item of the result of the second step is replicated along its

singular axes so as to make the array unishape.

The :~A **form** of an array A is an array of the same shape as A in which each item is replaced by its shape.

The following sequence shows the transformation of the form of an example array being trimmed:

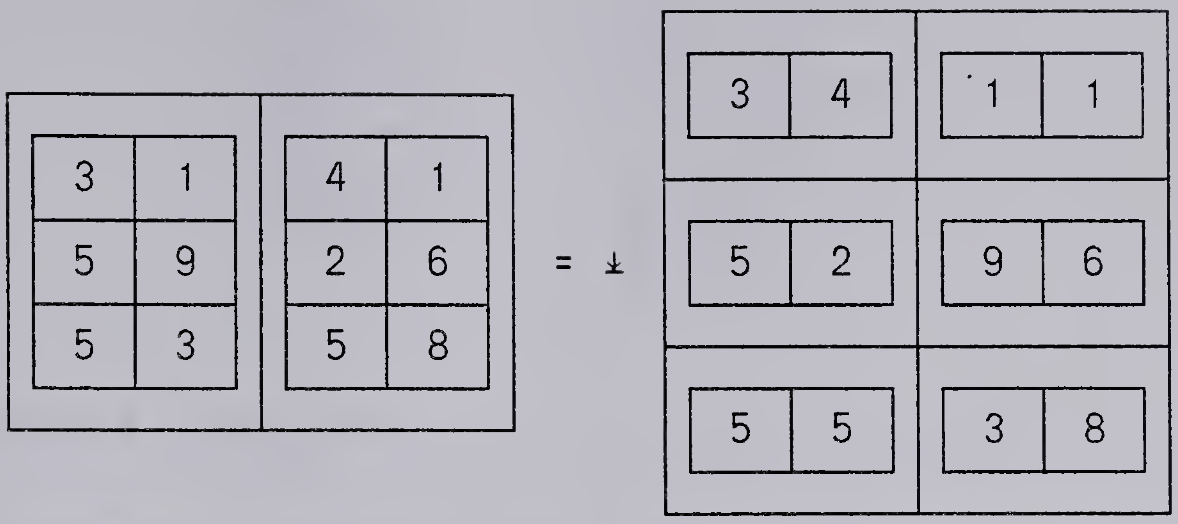
```
                [[3 4][3]θ[5 0 2]]                :~A
        [[3 4 1][3 1 1][1 1 1][5 0 2]]         step 1
        [[3 0 1][3 1 1][1 1 1][3 0 2]]         step 2
        [[3 0 2][3 0 2][3 0 2][3 0 2]]         step 3
```

The **packed** array ⤓A is obtained by first trimming A and then interchanging the top two levels of τA. As a result,

~⤓A = ~⊃τA and ~⊃⤓A = ~τA.

If, for example, A is a triple of pairs, then ⤓A is a pair of triples. The items in the first triple of ⤓A are, in main order, the first items in the pairs of A. The items in the second triple of ⤓A are, in main order, the second items in the pairs of A.

⤓[[1 2][3 4][5 6]] = [[1 3 5][2 4 6]].



In general, for any unishape array *U*, the array ⤓*U* is of shape ~⊃*U* and the items of ⤓*U* are of shape ~*U*. An item of ⤓*U* at address *I* holds, in main order, the items at address *I* of the items, in main order, of *U*. Consequently,

⤓ ≡ ι~⊃τω⊃̈±ω.

The effect of this definition can be seen in this example of ⤓[[1 2][3 4][5 6]]:

```
ι~⊃τ[[1 2][3 4][5 6]] ⊃̈± [[1 2][3 4][5 6]]
     ↔ ι~[1 2] ⊃̈± [[1 2][3 4][5 6]]
     ↔ [0 1] ⊃̈± [[1 2][3 4][5 6]]
     ↔ [[0 ± [1 2][3 4][5 6]] [1 ± [1 2][3 4][5 6]]]
     ↔ [[1 3 5][2 4 6]]
```

The following relationships hold for any array *A* and any unishape array *U*.

⊃⤓*U*=:⊃*U*
:⊃⤓*U*=⊃*U*
⊃⊃⤓*U*=⊃⊃*U*
⤓:⊃*U*=⊃:⤓⤓*U*
~⤓*U*=~⊃*U*
~⊃⤓*U*=~*U*

⊃∪*U*=⊃⊃*U*
\∪*U*=\⊃*U*
:\⤓*U*=⤓*U*
⤓:,*U*=,⤓*U*
⤓:⍮*U*=⍮⤓*U*
⤓⤓*A*=⍲*A*
⤓⤓⤓*A*=⤓*A*

- *Permeate Functions.*

A **permeate** function has the following properties. First, a permeate function is monadic. Second, a permeate function is defined for any argument which is a list of motes. Third, a permeate function ∆ extends to non-list arguments according to the relationship ∆*A* = :∆⤓,*A*. If *S* is a suit or list of motes and ∆ is a permeate function then ∆*S* is equal to :∆⤓,*S* which is equal to :∆⤓*S* which is equal to :∆∘*S* which is equal to ∘∆*S*. Therefore, the result of a permeate function applied to a suit or list of motes is always a mote. Any permeate function ∆ has a dyadic equivalent of the same symbol: ∆ ≡ ∆(α⍮ω).

- *The Arithmetic Functions.*

The +*A* **sum** of an array *A* is defined for lists of motes to be the sum of the items of the list. Any occurrence of ○ is considered to represent a zero, and any occurrence of a ⟂ is considered to represent a 1. If any item is not a valid number then the sum is ⍰. Sum is a permeate function. The dyadic equivalent of sum is called **plus**. An example of the operation of plus is

```
[2 3]ρ[0 1 2 3 4 5]+.[2 2]ρ[6 7 8 9]
+. [2 3]ρ[0 1 2 3 4 5]⍮.[2 2]ρ[6 7 8 9]
:+⍎,.[2 3]ρ[0 1 2 3 4 5]⍮.[2 2]ρ[6 7 8 9]
:+⍎.[2 3]ρ[0 1 2 3 4 5]⍮.[2 2]ρ[6 7 8 9]
:+. ⍳[2 2]⊃̈±[2 2]ρ[0 1 3 4]⍮.[2 2]ρ[6 7 8 9]
:+. [2 2]ρ[[0 0][0 1][1 0][1 1]]⊃̈± ...
:+. [2 2]ρ[[0 6][1 7][3 8][4 9]]
[2 2]ρ.+[0 6]⍮+[1 7],+[3 8],+[4 9]
[2 2]ρ[6 8 11 13]
```

The ×*A* **product** of an array *A* is defined for lists of motes to be the product of the items of the list. Any occurrence of ○ is considered to be a 0 and any occurrence of ⊥ is considered to be a 1. If any item is a zero then the product is zero, otherwise if any item is not a valid number then the product is ⍰. Product is a permeate function. The dyadic equivalent of product is called **times**.

The -*A* **negation** of an array *A* is a pervasive function. For any mote *M*, if *M* is a number then -*M* is the mote which is the additive inverse of *M*. *M*+-*M*=0. The mote ○ is considered to be 0 and the mote ⊥ is considered to be 1. If *M* is not a valid number then -*M* is ⍰. The dyadic equivalent of negate is called **subtract** and is defined as - ≡ α+-ω. The double negative --*A* converts any array *A* to strict numeric values in which all ○ map to 0, all ⊥ map to 1, and all characters map to ⍰.

The ÷*A* **reciprocal** of an array *A* is a pervasive function. For any mote *M*, if *M* is a number other then zero then ÷*M* is the mote which is the multiplicative inverse of *M*. *M*×÷*M*=1. The mote ○ is considered to be 0 and the mote ⊥ is

considered to be 1. If *M*=0 then ÷*M*=⍰. If *M* is not a valid number then ÷*M* is ⍰. The dyadic equivalent of reciprocal is called **divide** and is defined as ÷ ≡ α×÷ω.

- *The Predicate-Calculus Functions.*

The ¬*A* logical **not** of an array *A* is a pervasive function. For any mote *M*, ¬*M*=.○=*M*, except that ¬⍰=⍰. The double negative ¬¬*A* converts any array *A* to strict truth values in which each false augmented truth value is replaced by ○ and each true augmented truth value is replaced by ⊥.

The *A*≠*B* **inequality** of two arrays *A* and *B* is defined as:

≠ ≡ ¬α=ω.

The monadic predicate **unequal** is defined as:

≠ ≡ ¬=ω.

The ∧*A* **all** of an array *A* is defined for lists of motes. If any mote of the list is ⍰ then the value of ∧*A* is ⍰. Otherwise, if any mote of the list is ○ then the value of ∧*A* is ○. Otherwise, the value of ∧*A* is ⊥. All is a permeate function. The dyadic equivalent of all is **and**.

The ∨*A* **some** of an array *A* is defined for lists of motes to be ⊥ if any of the motes in the list are true augmented truth values. Otherwise, if any mote of the suit or list is ⍰ then the value of ∨*A* is ⍰. Otherwise, the value of ∨*A* is ○. Some is a permeate function. The dyadic equivalent of

some is **or**.

- *The Relational Functions.*

The ≥*A* predicate **increasing** is defined for unitype lists of motes. If *A* is not unitype or is all ⍰ then ≥*A* is ⍰. If 0=⊤⊃*A* then if there exists a sublist *S* of *A* such that ⊃*S* is numerically greater than ⊃⍏*S*, then ≥*A*=○, otherwise ≥*A*=⊥. Otherwise, if 'b̸'=⊤⊃*A* then if there exists a sublist *S* of *A* such that ⊃*A* is lexicographically greater than ⊃⍏*S*, then ≥*A*=○, otherwise ≥*A*=⊥. The predicate increasing is a permeate function. The dyadic equivalent of increasing is **greater or equal**.

The >*A* predicate **strictly increasing**, the ≤*A* predicate **decreasing**, and the predicate <*A* **strictly decreasing** are all defined analogously to the predicate increasing. The names of the dyadic counterparts are **greater**, **less or equal**, and **less** respectively.

- *Take and Scan.*

The *A*↑*B* **take** of an array *B* by an array *A* is an array of shape *A* in which each item of *A* specifies the number of items to be selected from *B* along the axis of *B* corresponding to the item in *A*:

↑ ≡ ια□ω.

If items are to be chosen along an axis that is shorter than

the corresponding value in *B* then the chosen items are, by the definition of ⊟, equal to \B.

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 4 & 5 \\ \hline \end{array} = [2\ 2] \uparrow \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array}$$

The ⌿Δ*A* **scan** of an array *A* is a list of shape #*A* in which the i'th item of the result is obtained by applying Δ to the list containg the first i items of ,*A*:

⌿ ≡ :ω̄. 1+ι#ω ⊃̈↑ ,ω

For example, ⌿+[*A B C D*] is equal to

:+[[*A*][*A B*][*A B C*][*A B C D*]]

- *The Cartesian Product.*

The ⊗*A* **cart** of an array *A* is an array of shape ∪:~*A*. Each item *I* of ⊗*A* is obtained from *A* by a two step process. First, the address of *I* in ⊗*A* is partitioned into sublists with lengths corresponding to the valencies of the items of *A* and this list of sub-lists is reshaped to the shape of *A*. Second, each of the items of the result of the first step is used as an address into the corresponding item in *A*.

For example, if *A* is a two by three table with form

:~*A* =. [2 3]ρ[[3 4][2 1][2 4][6 7][5 6 7][1 4]]

then

~⊗*A* = [3 4 2 1 2 4 6 7 5 6 7 1 4].

and

:(#~),*A* = [2 2 2 2 3 2]

The item at location [2 2 0 0 1 3 4 2 3 3 3 0 2] of ⊗*A*, for example, is determined by dividing that address up according to the 2 2 2 2 3 2 valencies of the items of *A*:

[2 3]ρ[[2 2][0 0][1 3][4 2][3 3 3][0 2]],

and using each of the items of this result as an address into the corresponding item of *A*. The [3 3 3] item at address [1 1], for example, would be used as an address for picking an item from the five by six by seven table at address [1 1] of *A*.

The above example illustrates the modularization that is used in the definition of the cart function. An example follows the definition.

The function

pa ≡ ~ω ρ. 0, ⌿+ :(#~) ω

applied to an array *A* returns an array of the same shape as *A*, in which an item at location *I* is the sum of the number of axes in each item of *A* that occurs, in ,*A*, before the item at location *I*.

The function

an ≡ pa ω + :ι. :(#~)ω

applied to an array *A* returns an array of the same shape as *A* in which an item at location *I* is a list of the numbers of the axes of the item at location *I* in *A*, where the axes are

numbered beginning at zero for the first axis of the first item of *A*.

The array ιυ:~*A* is an array of the same shape as ⊗*A* in which each item is its own address. The function

sa ≡ an ω ⊃̈⊂̈□ ιυ:~ω

applied to an array *A* yields an array of shape ~⊗*A* in which each item *I* is of the same shape as *A*, and contains items which are sub-lists of the address of *I*. Each of these sub-list items contains as many elements as there are axes in the corresponding item in *A*.

Now the cart function itself can be defined.

⊗ ≡ sa ω ⊃̈:□ ω

This is the array of shape ιυ:~*A* in which each item is of shape ~*A*, the items of which are the items of the items of *A* selected by the addresses in the items of sa *A*.

As an example, consider an array *A* which is a two by two table of three by four by five tables:

```
 #A = 4
 ~A = [2 2]
#~A = 2
```

:~*A* =

<table>
<tr><td><table><tr><td>3</td><td>4</td><td>5</td></tr></table></td><td><table><tr><td>3</td><td>4</td><td>5</td></tr></table></td></tr>
<tr><td><table><tr><td>3</td><td>4</td><td>5</td></tr></table></td><td><table><tr><td>3</td><td>4</td><td>5</td></tr></table></td></tr>
</table>

:(#~)*A* =

3	3
3	3

In this example, **pa** *A* is equal to

0	3
6	9

and **an** *A* is equal to

0 1 2	3 4 5
6 7 8	9 10 11

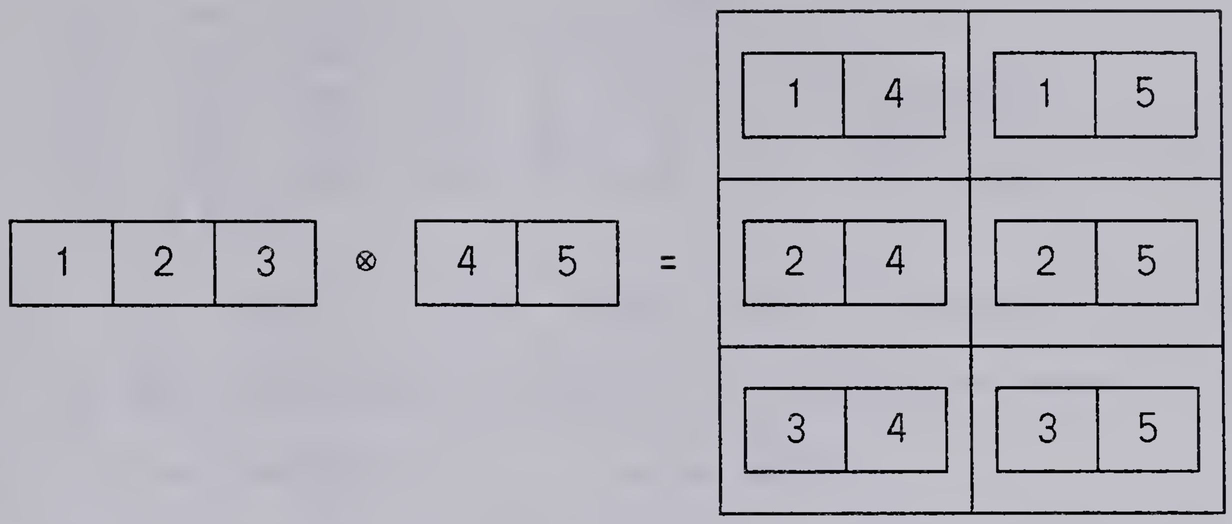
The item at location [0 1 2 1 2 3 2 3 4 0 2 4] of **sa** *A*, for example, will be

[2 2]ρ[[0 1 2][1 2 3][2 3 4][0 2 4]]

and so the items of the item at location [0 1 2 1 2 3 2 3 4 0 2 4] of ⊗*A* will be the item at location [0 1 2] of the item at location [0 0] of *A*, the item at location [1 2 3] of the item at location [0 1] of *A*, the item at location [2 3 4] of the item at location [1 0] of *A*, and the item at location [0 2 4] of the item at location [1 1] of *A*.

The dyadic *A*⊗*B* **cartesian** of an array *A* and an array *B* is defined to be

⊗ ≡ ⊗. α;ω.

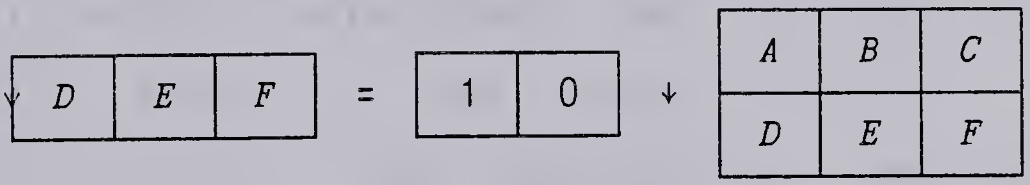


- *Drop, Mesh, Split, and Sections.*

The *A*↓*B* **drop** of an array *B* by an array *A* is obtained by deleting, along the *i*′th axis of *B*, the first *i*⊃*A* items:

↓ ≡ ι(~ω-α) ⊃̈+ α ▭ ω

```
[1 0]↓[2 3]ρ[A B C D E F]
        ←→ ι([2 3]-[1 0]) ⊃̈+ [1 0] ▭ ω
        ←→ ι[1 3] ⊃̈+ [1 0] ▭ ω
        ←→ [1 3]ρ[[0 0][0 1][0 2]] ⊃̈+ [1 0] ▭ ω
        ←→ [1 3]ρ[[1 0][1 1][1 2]] ▭ [A B C D E F]
        ←→ [1 3]ρ[D E F]
```



The *A*⋇*B*;*C* **mesh** of the pair *B*;*C* is a list of shape #*A* in which each ○ in *A* selects the next item from *B* and each ⊥ in *A* selects the next item from *C*:

⋇ ≡ #αρ. ∘⊃(⊃α⊃ω), ∤α ⋇ (⊃α=.0 1):↓ω

For example, [0 1 0 1]⋇[[*A B*][*C D*]] is equal to each of the following

```
4ρ. ∘A,. 1 0 1 ⋇ [[B][C D]]
4ρ. ∘A,. 3ρ. ∘C,. 0 1 ⋇ [[B][D]]
4ρ. ∘A,. 3ρ. ∘C,. 2ρ. ∘B,. 1 ⋇ [[][D]]
4ρ. ∘A,. 3ρ. ∘C,. 2ρ. ∘B,. 1ρ. ∘D,. θ ⋇ [[][]]
4ρ. ∘A,. 3ρ. ∘C,. 2ρ. ∘B,. 1ρ. ∘D,. 0ρ. ...
[A C B D].
```

The *A*↧*B* **split** of an array *B* by an array *A* is obtained from *B* by moving certain axes of orientation from *B* to the items of *B* according to *A*. For example,

```
0↧([2 3]ρ[A B C D E F]) =. [[A D][B E][C F]]

1↧([2 3]ρ[A B C D E F]) =. [[A B C][D E F]]
```

Each axis not named in *A* appears in the result, so that ¬(ι#~B∊A)\ι#~B =. ~(A↧B). Each axis named in *A* appears in the items of the result, so that ι#~B∊A\ι#~B =. ~⊃(A↧B). The definition of split is:

```
↧ ≡ :⊗. ι#~ω∊α ⊂̈⋇. ι(α‡~ω) ⊃̈⍮ :ι(α⌑~ω) ⌑ ω
```

For example, if A=[1 3] and ~B=[2 3 4 5 6] then A‡~B=[2 4 6] and A⌑~B=[3 5]. The array ι[2 4 6] is a two by four by six array in which each item is its own address. Call this array *P*. The array :ι[3 5] is equal to [[0 1 2][0 1 2 3 4]]. Call this array *Q*. The array P⊃̈⍮Q is a two by four by six array of pairs, each pair holding first its address in P⊃̈⍮Q and second the array *Q* itself. Call such a pair *R*.

Then, [∘⊥∘⊥∘]⊂̈⋇.P⊃̈⍮Q is a two by four by six array in which each item is [∘⊥∘⊥∘]⋇R. At location [1 2 4] for example, the value is

```
[[1][0 1 2][2][0 1 2 3 4][4]].
```

Each of these items are replaced by the cart of itself, the value at location [1 2 4], for example, becoming a three by five matrix which addresses the items of the three by five matrix at location 1 on the first axis, 2 on the third axis and 4 on the fifth axis. Each item of *A*↧*B* at address *I* will be the three by five array at address 1⊃*I* on the first axis, 2⊃*I* on the third axis, and 3⊃*I* on the fifth axis of *B*.

Each item of *A*↧*B* is a **section** of *B*. Intuitively, the item at address *I* in *A*↧*B* can be viewed as the section parallel to the axes of *B* named in *A* and normal to the other axes of *B* at the locations specified in *I*. The monadic split of an array is defined to be

↧ ≡ #~ω-1 ↧ ω.

If all the axes of *B* are named in *A* then there will be no axes in *A*↧*B*, so *A*↧*B* will be a single. On the other hand, all the axes will appear in the item of *A*↧*B*, but their orientation will be rearranged by permuting the axes according to *A*. Consequently, ⍳#~*A*↧*A*=∘*A*.

- *Transpose, Mix, and the Use of Sections.*

The *A*⍉*B* **transpose** of an array is obtained by reversing the axes order:

⍉ ≡ ⊃. #~ω - 1 - ⍳#~ω ↧ ω

The *A*⤒*B* mix of an array *A* by an array *B* is left-inverse to split in the sense that if *A* is a suit or list of distinct axis numbers for *B* then *B* =. *A*⤒.*A*⤓*B*. *B* is first trimmed, and then the axes of the items of *B* that are named in *A* are moved up to become axes of the result itself. The monadic mix of an array is defined to be

⤒ ≡ #~ω + ι#~⊥ω ⤒ ω.

For example, the value of 1 ⊂̈↓ 1⤓[2 3]ρ[*A* *B* *C* *D* *E* *F*] is [[*B* *C*][*E* *F*]], and the value of 1 ⤒ [[*B* *C*][*E* *F*]] is [2 2]ρ[*B* *C* *E* *F*].

For many functions, the mix of the function applied to the split of an object has the effect of applying the function to all the sections selected by the split arguments. Variations of the functions can be written to accept, instead of the usual arguments, an argument specifying the usual argument and the axis application information. For example, the sublist function can accept as left argument a pair specifying both the selection mask and the application axes. The new definition (in terms of the old one) might be

⊃⊂α⤒. ∘⊃α:\(⊂α⤓ω).

Pervasive functions can be applied to slices with the expression :Δ(*A*⤓*B*).

Notice that the transpose function can in fact be defined in terms of these re-axising operations:

⍉ ≡ ⤒⤓↧ω.

- *Supplant, Rendering, and Canonical.*

The *A*⍮*B*±*C* **supplant** of an array *C* by a pair *A*⍮*B* is the same as *C* except as follows. *A* specifies an address in *C*. If the address is valid, then the item of *C* at that address is supplanted by *B*.

± ≡ ~ωρ. ι#~ω ⊃̈= ⊃α ⋇. ⊃α∗⃒,ω ⍮. ⊂α.

For example, [2 5]±ι5 is equal to [0 1 5 3 4].

The ∇*A* **rendering** of an array *A* is primitive to the interpreting system. Whereas the interpret function ⌶ returns the value of its argument treated as an expression, the ∇ function returns the function represented by its agrument. If *A* is not a valid function representation, then the value of ∇*A* is the everywhere-⍰ function ⊃(⍰⍮ω) which always returns the value ⍰. If the argument is of the form '**n**≡*e*' then the name **n** may appear in the expression *e* to effect a recursive definition.

The ∇̍∆ **canonical** representation of a function ∆ returns a nil-adic function which is equal to the object which represents the definition of the function ∆. If ∆ is primitive then ∇̍∆='∆'.

This section has illustrated the defintion of a number of complex functions using only the array theory primitives, mote transformation functions, and the primitives of the interpreting system. Given this foundation, many more useful functions can be defined, and the encoding of the redundancy in a solution can be represented in a functional way.

3.4: Postscript to Chapter Three

The theory of arrays is not bound to the syntax used here for its exposition. These formal expression are simply convenient for the level of description at which we are working. It is far more important that the representation is a functional one in which every expression returns a result without the use of transfer of control, and no expression produces side effects. The lambda calculus of Church and the FFP notation of Backus are two alternate systems for the formalization of functions, although they have explictly been avoided because of their unnecessary complexity.

The notions of the type and the prototype of an empty array, although perhaps initially disturbing, are formally simple and seem to work well in the definition of complex functions. Although other schemes would work, the prototype appears to be an elegant idea. In the definition of the

locate function, for example, an empty right argument results in an empty list of addresses, just as one would expect.

As well, motes are a notion that seems to be effective, even though the concept is not straightforward. En-mote and de-mote functions for converting strings to motes and motes to strings might be interesting since they would allow the extension of the notion of type. The justification for the multiplicity of empty arrays and for the self-nesting of motes lies in their support of the axiomatization of the theory. The importance of the elimination of special cases from the axioms, such as is required for the equivalent of motes and empty arrays in other theories, cannot be overemphasized. Without this uniformity, the number of cases to be examined in a proof grows exponentially, with the uniformity, the proof remains linear.

Many of the recursive definitions terminate on one of the following two conditions. The application of a function to the items of a mote is equivalent to the application of the function to the mote itself. Pervasive functions, therefore, stop at motes. The union of a single with the empty reshaping of any array is the item of the single: $\circ A \cup \backslash B = A$. Therefore, any function which can compute the number of items in its result can select just that many items, and can therefore terminate at an expression which unites a single to the empty reshaping of the recursive

application of the function. (See the definition of the mesh function for an example.) These boundry conditions are more uniform than those established arbitrarily by a conditional expression.

The power of the primitives and the data structure can be seen in the ease of definition of the more complicated functions in section 3.3. The definition of pack, for example, is based on numerate and project, which is based on box, which is based on pick, which is based on sublist. The each-transforms used to build the hierarchy are themselves built only of the shape, reshape, sublist, pair, and union primitives and the axioms which allow the termination of recursion. Even for functions as complicated as cart, an understanding of the simple primitives and the hierarchical development of the definition is all that is required to completely understand its effect.

The each and replacement transforms, in particular the each-left and each-right transforms (which are an extension of More's work developed in this experiment), can be seen to be a particularly powerful tool in the development of algorithms based on generalized arrays. This effect is a most interesting result of the theory.

As a further example of this hierarchical combination of the items of one object with the items of another object, study the definition of a function similar to *APL*'s

outer-product function:

op ≡ ⊥̵ω⊂̈⊃̈ω̄α

This function builds a table of the results of the argument function ω̄ applied to each pairing of the elements of α and ω.

The overriding consideration for extension of the interpretation of the axioms, for the introduction of new axioms, and for the definition of new functions, is the maintenance of the theory as closed, standard, one-sorted, and axiomatic. These properties ensure the minimization of the apparent complexity of representations.

Chapter Four: Implications

> *"When the mind grapples with a great and intricate problem, it makes its advances step by step, but with little realization of the gains it has made, until suddenly, with an effect of abrupt illumination, it realizes its victory."*
>
> H. G. Wells

In Chapter Four the system model established in Chapter Two will be expanded upon. As well, the results of the thesis are reviewed and certain areas of extension are outlined.

4.1: Systems Considerations

The motivation for this high-level bootstrapping process was established to be the attainment of a very-high-level language tailored for the problem-solver. In order to fully grasp the potential of the functional, generalized array approach, the view of the rest of the system as seen by a processor must be understood, for it is the powerful support system that completes the usefulness this approach.

This discussion is, however, not to be considered as a proposal. Many alternate formulations of the system can be made. The particular formulation below is intended to reveal a systems philosophy, within which it is intended that any particular system should operate.

The system proposed here is tailored for a development environment. The system is not designed for large scale production computing. Increasingly though, the problems encountered during the development of a large scale production system are more difficult than the those encountered during its production life. The success of systems such as the UNIX/PWB[1] programmers workbench provide a basis for the continued discussion of this development environment.

In the development environment, the efficiency of the scheduler's algorithm and the ease of use of the system by the naive user are far less important than the power of the system from the point of view of the experienced user requiring a work-bench. The experienced user appreciates elegance, simplicity, clarity, and generality. Most often, the software generated by him will not be used for long periods of time and will not be required to perform computationally difficult tasks, at least in the breadboard stage.

[1] Dolotta, T. A., Haight, R. C., and Mashey, J. R., "The Programmer's Workbench," *The Bell System Technical Journal 57 (Part 2)*, 6(July-August 1978):2177-2200.

- *The Shared-Variable Model.*

As shown in figure 2.3-1, the underlying model for our discussion of systems considerations is the shared-variable system. A shared variable system provides three services:

1 ∘ Support of an interpreter capable of determining the meaning, in terms of a result object, of an argument to the interpret (ɪ) function. Inherent in this ability is, of course, the knowledge of all the functions used in the expression presented as the argument.

2 ∘ Support of a scheduler which manages the allocation of the processors of the system (and interleaves the tasks in the time-sharing case).

3 ∘ Support of a shared-variable processor capable of switching generalized arrays between processes. The interpret function obtains a processor capable of the interpretaion (the one it is running on, for example) and waits for the result. The scheduler is accessed by the interpreter, for the purpose of parallel task control, via the shared-variable processor.

The effects of the interpreter are those discussed in the previous chapter: the interpretation of the strings representing expressions and of their operation on generalized arrays.

The scheduler is largely unimportant from the point of the developer, and of course, should therefore not be of concern to him. Nevertheless, there are a few scheduler functions that he may need directly. Since the scheduler and the shared-variable supporter are not ignorant of each other, the user can communicate with the scheduler via the same shared variable mechanism that is used for all other communications.

The shared-variable support rests on a few functions which are primitive to the support system, as described below. These functions parallel those discussed by Lathwell[1] but include explicit communication functions since there is no assignment operation in which to hide the support.

Communication with another processor is established with a **shared-variable offer** via the ⌼ function. In order for a processor (*ME* for example) to tender the communication-interface offer, a suit of quintuples must be established as the argument for the ⌼ primitive. The first item of each quintuple should be the string '*TENDER*'. The second item of each quintuple should be the string that denotes the processor that the offer is being tendered to,

[1] Lathwell, "System Formulation and *APL* Shared Variables," p. 356-357.

'*YOU*' for example. The third item should be a string which will denote this particular offer and which will be required for the discussion of the offer by other shared variable functions, '*MSG*' for example. The fourth item should be the access control vector (referred to as the ACV) discussed below. The fifth item should be either '*WAIT*' or '*FREE*' depending on whether the system is to return from the offer as soon as it is made or wait for the offer to be matched.

A general offer, to any processor, may be made by specifying the nil processor name ⍬.

The value of ⌼*A* is a suit with either ○ to indicate that the corresponding offer has not been matched or ⊥ to indicate that it has.

The **access-control vector** can consist of any of the following characters:

"*R*" to indicate that two successive receptions by *ME* require an intervening transmission by *YOU*.

"*T*" to indicate that two successive transmissions by *ME* require an intervening reception by *YOU*.

"$\boxed{R}$" to indicate that two successive receptions by *YOU* require an intervening transmission by *ME*.

"$\boxed{T}$" to indicate that two successive transmissions by *YOU* require an intervening reception by *ME*.

The actual access-control state is the logical sum of these constraints. If a constraint occurs in the access-control vector as specified by either user then the constraint occurs in the access-control state. When the access-control state requires a particular interleaving of the communication, a transmission or reception may be required to wait until the other processor in the share has made the appropriate access. Initially, there will be no constraints imposed except those specified in the offer.

The ⌼ function can be used to specify a change in the constraints, by specifying an argument like that of the offer but with the following difference. The first item of the quadruple should be the string '*CONTROL*'. The remaining items are the same. The result items are either ⊥ to indicate that the control has been changed or ∘ to indicate that it has not. If the fourth item of the quintuple is ⏻ then the access state is reset to no constraints, otherwise the new constraints are added to the existing ones.

The ⌼ function can be used to disconnect an interface. In this case the argument items are triples consisting of the string '*RETRACT*', the processor name, and the share name. The result items logically indicate the success of the retractions.

The ⌼ function can be used to inquire into the access-control state status. The argument items are triples consisting of the string '*QUERY*', followed by the processor name and the share name. The result items are pairs holding the access-control-state string described above, and the name of the processor offering the match (⌽ if unmatched).

The ⍈ **transmit** function accepts a suit of triples consisting of the name of the processor, the name of the share, and the value to be transmitted. The items of the value of the function logically indicate the success of the transmissions.

The ⍇ **receive** function accepts a suit of pairs consisting of the name of the processor, and the name of the share. The items of the result are the values of the shared arrays.

- *The Interface to the Scheduler.*

Communication with the scheduler is accomplished via the special shared-variable ⧆. The argument to ⧆ is, like ⌼, a suit of suits, and the result is a suit in which the items correspond to the argument items.

If an argument item is a triple consisting of the string '*CONCEIVE*', a processor name, and an expression encoded as a string, then a process of the indicated name will be created to interpret the indicated expression. The result item will

logically indicate the success of the conception.

If an argument item is a pair consisting of the string *'WAIT'* and the processor name, then the current process will be made to wait for the named process to terminate. The result item will be the value of the expression executed by the process. Once a process has successfully waited for the completion of a process, the completed process and the value of its expression will be forgotten.

Other coordination commands such as the die and wound directives may be similarly provided[1]

If an argument item is a pair consisting of the string *'QUERY'* and a processor name then the result will be a string indicating the status of the processor: *'RUN'*, *'WAIT'*, or *'DEAD'*, for example. If the result item is ⌽ then the named process is unknown.

- *The History Process Interface.*

As was stated in Chapter Two, the entire dynamic environment could theoretically be passed about between the

[1] Easwaran, K. P., Gray, J. N., et al, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM 19*, 11(1976):624-633.

executing functions (as proposed by Backus[1]). Alternatively, the shared variable system can support one or more history processes, somewhat akin to the file systems of the classical operating-systems. An example of such a history process follows.

The history process is accessed via a system assigned name such as '*MEMORY*'. A shared-variable interface must be established to the history process, after which the receive and transmit functions may be used to store and retreive arrays. The transmission of a triple consisting of the string '*STORE*', a name, and a value, remembers the value as being associated with the name. The transmission of a pair consisting of the string '*FETCH*' and a name retrieves the value associated with the name.

The history function keeps a suit of pairs consisting of names and values. A store request unites a new name with specified value to the suit if the name is not in the replacement extract of the list. If the name is there then the corresponding item is supplanted by the name paired with the new value.

[1] Backus, *"Can Programming Be Liberated...,"* p. 74-88.

A retreive request simply looks for the name in the replacement-extract of the history suit and returns the corresponding value (or ⍰ if no name is matched). The completion of an operation by the history process is followed by its reinvocation of itself passing as its new argument the new history suit.

On the death of a process, all the history items associated with the process can be forgotten. Certain items can, however, be identified to the history process as archived items in which case they will remain until explicitly deleted via such an imperative.

Of course, the history process can be arbitrairly more sophisticated than the one just sketched. In particular, a hierarchical directory structure such as the UNIX file system[1] would be easily implementable.

The conceptual view of such a history process is as a **name space**. In a name-space environment, certain functions which are not local to the name-space of a process can be identified by their name and name-space identification. A user-defined addition function, "+" for example, could use the primitive sum function by explicitly identifying the

[1] Ritchie, D. M., "The UNIX Time-Sharing System: A Retrospective," *The Bell System Technical Journal* 57, 6(July-August 1978):1953-1954.

name-space of the symbol when it is used. Unless otherwise specified, the current name-space is always assumed to be the domain of discourse.

Now it is possible for libraries of functions to be established in name-spaces in order to allow their repeated use. By convention, the interpreter can consider a name consisting of a left-part followed by the reference-symbol shriek (!) followed by a right-part to be a reference to the function named by the right-part in the name-space named by the left-part.

During the creation of a process for a user's terminal, the system can ask the history process for the archived initial-expression to be executed at the terminal. If the user has not specified an initial-expression then a default such as *SYSTEM!TERMINAL* could be used.

The detection of an interrupt by the interpreter could cause the interpreter to have the scheduler suspend the interrupted process and schedule an interrupt-handler named in the history-process. The interrupt handler can, of course, control the interrupted process via the scheduler.

The defined functions of the system can now exist as strings remembered by a history process. Both the interpreter and the user can access these strings, and the interpreter can apply them with the render primitive. Consequently, the user can access the utilities of his

choice for the management of the data and defined functions of his name-space.

- *The Services of the Interpreter.*

The interpreter will, in addition to performing the interpretations required by Chapter Three, provide the following services.

If the stored representation is not a suit of characters then a single union will be performed. This allows the user to save his functions as a suit of strings for ease of comprehension.

Any characters occurring between matched illumination symbols (⍝) will be deleted. This allows documentary comments to be included in the source code.

The conventions supported by the interpreter such as the conditional notation and the name-space references will be converted to the appropriate function invocations.

The conditional notation *P*→*Q*;*R* will be converted into an expression equivalent to ⌶*P*⊃.'*Q*'⍪'*R*'. Alternate conditional expressions, such as the parallel evaluation of a predicate set followed by the parallel evaluation of the consequent sub-listing, could, by convention, be similarly supported.

This support system is general enough to allow the conceptualization of a complete development environment. As experience is gained in the use of the system, many of the high-level actions described herein may be replaced by more efficient low-level implementations, but the high-level model will remain as the conceptual view of the system.

4.2: Review and Conclusions.

Many of the conventional programming languages today attempt to provide the user with all the possible features that he could require. As a result, these languages have become gargantuan monsters, capable of devouring a student in a single pass. Alternately, Iverson's[1] philosophy of providing the minimum necessary features and a very powerful tool for their combination can prevent unchecked growth of languages. Perhaps more powerful primitives will be proposed someday, but the features provided by the behemoths have clearly outstripped the needs of the user.

Modern procedural languages are too difficult to understand, too difficult to put together, and too difficult to repair after one of their too frequent breakdowns. Instead of encouraging the haphazard specification of an

[1] Iverson, K. E., personal conversation, February 1980.

algorithm via explicit but arbitrary boundary conditions and termination criteria, the functional array-processing methodology requires careful thought before a solution can be expressed with the notation.

Moreover, the functional methodology results in the specification of a general algorithm, one in which the boundary conditions are determined by the structure of the data and not by a set of arbitrary values. The use of control structures and the consequently complex state-transformation description of the solution is largely avoided by the functional manipulation of an aggregate data-structure. The item-wise transformation of the data by functions written for single-item operation allows the simple description of the solution in terms of operations on the data.

Largely, this power is due to the ability of the generalized arrays to support the properties of a data-structure that allows the easy conceptualization of the effects of operations on the data. The number of accessible complexity components required for the comprehension of an expression is reduced to the knowledge of the structure of the data and the effect of the primitives on the data.

Furthermore, expressions and the proof of the meaning of an expression are greatly simplified with the aid of the theorems of the functional language. If a particular

composition of a set of functions can be located in a table of compositions, and an equivalent but simpler composition can be found there, then the simpler compositon can be used in place of the more complex composition. The simpler composition may even provide conceptual insight into the solution being represented.

These principles apply to the operators as well as the functions of the language, so that there are no control strutures that need be explicitly accounted for in the proofs of algorithm correctness.

Many different generalized-array axiomatizations and interpretations of generalized-array axiomatizations are possible. For example, instead of the present hieroglyphic notation that proves so convenient for the abstract black-board discussion of the axiomatization, a keyword or even context-directed notation could be used. As well, the notion of type as the typical member of a family could be extended to allow something like user-defined types. Classes of user-defined motes could have typical items different from those of the existing archetypes.

The extension of boolean values to the integers zero and one and the extension of the notion of boolean values to typical versus non-typical items is simply a convenience. The relationship specifying that a milliliter of water is equal to a cubic centimeter of water and weighs one gram is

another example of a particularly conventient relationship.

Perhaps the individuals of the theory should be n-dimensional numbers. Complex numbers would then be just two-dimensional motes. Real numbers would be complex numbers with no complex part. Integers would be real numbers with no fractional part. Characters could just be integer indicies into character generating devices.

Before any of these extensions are made, as always, the power of the additional features must be weighed against their practicality. Instead of introducing complex or rational numbers as motes (with the required associated axioms, of course), they can simply be represented as a pair of real and immaginary parts or numerator and denominator values. As the usage patterns stabilize, certain conventions or standards may emerge, but it is not necessary to pre-specify these standards in the formal system.

This same tradeoff arises in the discussion of the tagging of arrays. Currently, the addresses of the arrays are ordinals. Alternately, a mapping between other addresses and these ordinals could be supported by the system so that general associative addressing can be performed on axis-location names. A rainfall table could, for example, have one axis addressed with the numbers 1970 through 1980 and the other axis addressed with the strings *'JAN'* through *'DEC'*. In order to support such a scheme, an

axiomatization would be required. Alternately, the mappings can be supported by table driven mapping functions, and for this conceptualization, they are.

A similar tradeoff arises during the consideration of the modifier problem.[1] Certain functions have very closely related relatives, such as the structural functions on different axes. Perhaps these related functions could be represented by a family name and a modifier. On the other hand, functional versions of these modifiers can be used, and for this conceptualization, they are.

The functions introduced in Chapter Three do not produce error messages. Instead, they always return an array, although its value may be ⍰. The suggestion that the ⍰ should have a LISP-like property list on which a function could label a detected error has been entertained. Alternately, functions which could generate error messages can be designed to return a pair consisting of the status of the result and the value of the result. In this conceptualization, they are.

[1] Iverson, K. E., "Operators and Functions," *Research Report RC7091 (#30399)*, (San Jose: IBM Research Division, 1978), p. 13-15.

Clearly, there is room for expansion on these foundations. The intent herein is not, however, to provide these expansions, but rather to provide a strong enough foundation for the study of the expansion itself. The foundations of the procedural languages were certainly strong enough for some time, but they are beginning to crumble under the magnitude of the algorithms currently being represented.

Backus has suggested that it is the very procedural languages themselves that have caused the stagnation of innovative hardware design[1] The procedural accent learned with the procedural languages prevents designers from attempting a more functional solution to their programs. Perhaps, it is felt, the exposure of functional solutions to more students at earlier phases in their studies will prevent that accent from being acquired and will eventually lead to innovative hardware configurations not centered around the von Neumann bottleneck.

In addition to changes in the hardware itself, advances in automatic optimization allow functional solutions to be optimized as they are interpreted. For example, the composition of two functions can be replaced by a single

[1] Backus, J. W., "Can Programming be Liberated...," p. 88-90.

function which accepts, in addition to its arguments, the address of a particular item which is required. In this way, items that are never used in a calculation never need be calculated.

In *APL*, for example, the index function ι does not actually generate the set of integers from one to its argument. Instead, it generates a triple to describe an arithmetic progression vector: the first item, incriment, and number of items in the vector. In this way addition of a constant, multiplication by a constant, and the deletion of leading elements can be handled by simply altering the triple. Consequently, the *APL* expression $(\alpha-1)\downarrow\iota\omega$ yields the integers from α to ω, independent of their magnitude, without actually generating $\iota\omega$ values. The expression in conceptually simple, and the automatic optimization of its implementation yields a computationally simple result.

If a notational framework provides no machinery for explicitly indicating the delayed evaluation of the arguments of a function, the framework is inadequate! For this reason, Church's lambda calculus delays the evaluation of all its arguments until after substitution. The

[1] Wegner, P., *Programming Languages, Information Structures, and Machine Organization*, ed. R. W. Hamming and E. A. Feigenbaum, (New York: McGraw-Hill Book Co., 1968), p. 187.

functional framework developed herein prevents this problem by allowing evaluation before substution (the case for functions) or after substution (the case for operators).

Although the promotion of memory-less execution is usually not respected, we can see that the provision of a message-switching system allows historic-information access to be treated just like the input-output operation it is. Eventually, as the memory-access bottleneck is relieved, the access to historic information will probably become an inter-process message problem anyway.

The author has recently completed a commercial applicaton using a functional form of *APL*. A translation system maintains the functional text and generates the equivalent *APL* functions. The translation system is written in iteself. A project planning tool of about the same magnitude as the translation system was then developed. Together, this effort represents about two hundred *APL* functions averaging about eight lines translated from twelve. This represents about 1600 *APL* expressions.

The planning tool took the author only a few man-weeks to complete. The customer obtained the system quickly. The system was not expensive to develop and, in the planning environment, is not expensive to operate. The system is very friendly to the user. No module is more than a dozen or two lines, including all documentation. And of course,

the system is completely modular and heirarchical.

This planning tool illustrates the feasability of the functional approach. The largest problems encountered in its development were the data-structure and operator limitations imposed by *APL*. These are just the problems that have been addressed in this thesis.

This functional notation for handling generalized arrays is, as can be seen from the above discussion, only in its infancy. Nevertheless, the results that are being obtained are clear indications that many of the notions which More and Backus have captured in their inspirations are sound.

When arguments were originally being made for the reduction of the use of explicit branching in procedural languages, opponents of the proposal vigorously protested the immagined loss of brevity in representation and efficency in execution. Experience has shown, however, that without the complexity reduction that accompanies the branching reduction, large representations cannot be comprehended at all.

Opponents of the notions developed in this thesis present a similar argument when faced with the loss of their familiar procedural tools, but their concern is misplaced for the same reasons.

> The practice of first developing a clear and precise definition of a process without regard to efficiency, and then using it as a guide and a test in exploring equivalent processes possessing other characteristics, such as greater efficiency, is very common in mathematics. It is a very fruitful practice which should not be blighted by premature emphasis on efficiency in compter execution[1].

There will always be room for any technique that can be used to solve a problem. Situations which need the fineness of description characteristic of the procedural languages and item-wise data structures may always exist, but there is no doubt that situations which require the power of the generalized array and the functional decriptions do now exist.

In particular, a procedural form tends to restrict ones intuitive grasp of a problem, and so is particularly poor for the solution of complex problems and for the teaching of the concepts of computing science. A procedural solution should only be attempted once the performance of a functional solution shows its necessity. A procedural model should only be taught to students who have grasped the concepts as exposed by a functional model.

[1] Iverson, K. E., "Notation as a Tool of Thought," 1979 ACM Turing Award Lecture, *Communications of the ACM 23*, 8(August 1980):444-465

Bibliography

Backus, J. W., "Can Programming be Liberated from Its von Neumann Style? A Functional Style and Its Algebra of Programs," 1978 ACM Turing Award Lecture, *Research Report RJ2234(30357)4/25/78*, (San Jose: IBM Research Laboratory, 1978)

Berke, P., "Data Design with Array Theory," *Technical Report G320-2123*, (Cambridge, IBM Scientific Center, July 1978)

Berke, P., "Tables, Files, and Relations in Array Theory," *Technical Report G320-2122*, (Cambridge, IBM Scientific Center, July 1978)

Burge, W. H., *Recursive Programming Techniques*, (Reading, Mass.: Addison Wesley Inc., 1975)

Church, A., *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies, Number Six, (Princeton: Princeton University Press, 1941)

Dolotta, T. A., Haight, R. C., and Mashey, J. R., "The Programmer's Workbench," *The Bell System Technical Journal 57 (Part 2)*, 6(July-August 1978)

Easwaran, K. P., Gray, J. N., et al, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM 19*, 11(1976)

Ghandour, Z. and Mezei, J., "Generalized Arrays, Operators and Functions," *IBM Journal of Research and Development 17*, 4(1973)

Gull, W. E. and Jenkins, M. A., "Recursive Data Structures In *APL*," *Communications of the ACM 22*, 2(January 1979)

Haney, Frederick M., "Module Connection Analysis—A Tool for Scheduling Software Debugging Activities," in the Fall Joint Computer Conference, (Montvale, N. J.: AFIPS Press, 1972)

Hofstadter, D. R., *Gödel, Escher, Bach: an Eternal Golden Braid*, (New York: Basic Books Inc., 1979)

Honig, W. L. and Carlson, C. R., "Toward An Understanding Of (Actual) Data Structures," *The Computer Journal 21*, 2(May 1978)

Iverson, K. E., "Notation as a Tool of Thought," 1979 ACM Turing Award Lecture, *Communications of the ACM 23*, 8(August 1980)

Iverson, K. E., "Operators and Functions," *Research Report RC7091 (#30399)*, (San Jose: IBM Research Division, 1978)

Iverson, K. E., "The Role of Operators in *APL*," *ACM-STAPL/SIGPLAN Proceedings APL79 Conference*, (May 1979)

Kleene, S. C., *Introduction to Metamathematics*, (Amsterdam: North-Holland Publishing Co., 1952)

Lathwell, R. H., "System Formulation and *APL* Shared Variables," *IBM Journal of Research and Development 17*, 4(July 1970)

Leavenworth, B. M., and Sammet, J. E., "An Overview of Nonprocedural Languages," in Proceedings of a Symposium on Very High Level Languages, *ACM SIGPLAN Notices 9*, 4(April 1974)

McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1," *Communciations of the ACM 3*, 4(April 1960)

Miller, George A., "The Magical Number Seven, Plus or Minus Two: Some limits on Our Capacity for Processing Information," *The Psychological Review 63*, 2(March 1956)

Miller, James G., *Living Systems*, (New York: McGraw-Hill Book Company, 1978)

Minsky, M., *Computation: Finite and Infinite Machines*, (London: Prentice-Hall International, Inc., 1972)

More, T., "A Theory of Arrays with Applications to Data Bases," *Technical Report G320-2106*, (Philadelphia, IBM Scientific Center, May 1973)

More, T., "An Interactive Method for Algebraic Proofs," *Technical Report 320-3005*, (Philadelphia, IBM Scientific Center, September 1971)

More, T., "Axioms and Theorems for a Theory of Arrays," *IBM Journal of Research and Development 17*, 2(March 1973)

More, T., "Nested Rectangular Arrays for Measures, Addresses, and Paths," *ACM-STAPL/SIGPLAN Proceedings APL79 Conference*, (May 1979)

More, T., "Notes on the Axioms for a Theory of Arrays," *Technical Report G320-3017*, (Philadelphia: IBM Scientific Center, May 1973)

More, T., "Notes on the Development of a Theory of Arrays," *Technical Report 320-3016*, (Philadelphia: IBM Scientific Center, May 1978)

More, T., "On the Composition of Array-Theoretic Operations," *Technical Report G320-2113*, (Cambridge: IBM Scientific Center, May 1976)

More, T., "The Nested Rectangular Array as a Model of Data," *ACM-STAPL/SIGPLAN Proceedings APL79 Conference*, (May 1979)

More, T., "Types and Prototypes an a Theory of Arrays," *Technical Report 320-2112*, (Cambridge, IBM Scientific Center, May 1976)

Post, Emil., "Formal Reductions of the General Combinatorial Decision Problem," *American Journal of Mathematics*, 65(1943)

Quine, W. V., "Unification of Universes in Set Theory," *Journal of Symbolic Logic 21*, (1956)

Ritchie, D. M., "The UNIX Time-Sharing System: A Retrospective," *The Bell System Technical Journal 57*, 6(July-August 1978)

Schwartz, Jacob T., *On Programming: An Interm Report on the SETL Project*, (New York: Courant Institute of Mathematical Science, revised 1975)

Simon, Herbert A., *The Sciences of the Artificial*, (Cambridge, Mass.: The M.I.T. Press, 1969)

Suppes, P., *Axiomatic Set Theory*, (New York: Dover Publications Inc., 1972)

Wegner, P., *Programming Languages, Information Structures, and Machine Organization*, ed. R. W. Hamming and E. A. Feigenbaum, (New York: McGraw-Hill Book Co., 1968)

Wiedman, Clark, "*APL* Problems with Order of Execution," University Computing Center, Graduate Research Center, University of Massachusetts, Amherst, Mass.

Winograd, Terry, "Beyond Programming Languages," *Communciations of the ACM 22*, 7(July 1979)

Wulf, W. A., "Some Thoughts on the Next Generation of Programming Languages," in *Perspectives of Computer Science*, ed. A. K. Jones, (New York: Academic Press, 1977)